REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE: Sep 94

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
Using Machine Learning to derive efficient cost+ Performence estimates in VLSI CAD Designs

5. FUNDING NUMBERS

6. AUTHOR(S)
Donld S. Gelosh

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT Students Attending:

University of Pittsburgh

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/CI/CIA
94-038D

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DEPRTMENT OF THE AIR FORCE
AFIT/CI
2950 P STREET
WRIGHT-PATTERSON AFB OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release IAW 190-1
Distribution Unlimited
MICHAEL M. BRICKER, SMSgt, USAF
Chief Administration

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

DTIC ELECTE DEC 14 1994 B

19941207 065

DTIC QUALITY INSPECTED 1

14. SUBJECT TERMS

15. NUMBER OF PAGES
180

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

94-038 D

USING MACHINE LEARNING TO DERIVE EFFICIENT COST AND PERFORMANCE ESTIMATES IN VLSI CAD DESIGNS

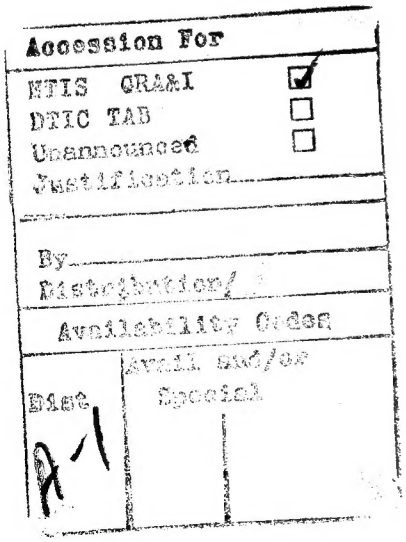
by

Donald S. Gelosh

B.S. in E.E., The Ohio State University, 1981
M.S. in Computer System Design,
The University of Houston at Clear Lake, 1989

Submitted to the Graduate Faculty
of the School of Engineering
in partial fulfillment of
the requirement for the degree of
Doctor
of
Philosophy

University of Pittsburgh
1994



The Author grants permission
to reproduce single copies.

Signed

COMMITTEE SIGNATURE PAGE

This dissertation was presented by
Donald S. Gelosh

It was defended on
September 9, 1994

and approved by

Dorothy E. Setliff, Assistant Professor
Committee Chairperson

Bruce G. Buchanan, Professor
Committee Member

Marlin H. Mickle, Professor
Committee Member

J. Thomas Cain, Associate Professor
Committee Member

Steven P. Levitan, Associate Professor
Committee Member

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Dr. Dottie Setliff, for her patience and willingness to go the extra mile by proofreading this thesis over weekends and during her vacation time. I could not have completed it without her determination and devotion to this research. Over the past three years, we have had several thought provoking discussions about this research and each one produced a new idea or new direction. I wish her luck in her future endeavors.

Next, I would like to thank Jiyong Ahn who, in my opinion, is a wizard at the computer keyboard. He has helped me in numerous ways to get through this research. He is a devoted friend who has never let me down and has always offered a bit of encouragement or humor to get me through the tough spots. I also wish him luck as he starts on the road to his Ph.D.

I would like to thank the members of my committee, Dr. Steve Levitan, Dr. J. Thomas Cain, Dr. Marlin Mickle, and Dr. Bruce Buchanan. Throughout this research, they have shown a lot of patience and understanding. I would especially like to thank Dr. Buchanan for putting me on the road to using a machine learning system as a data analysis tool. This suggestion solved a very difficult problem and allowed me to reach the goals of this research. I would also like to thank Dr. Levitan for lending me his knowldege of the various CAD tools and especially for his support in the way of a guest account on his workstations. I could not have finished this research without his help.

I would also like to thank some of the other graduate students, Yee-Wing Hsieh, Kyu-Myung Choi, and Steve Frezza for the help they've given me in understanding the various VLSI CAD tools. I would also like to thank them for being understanding during long runs of the layout tool which slowed down everyone's work. Without their guidance and patience, I would still be fighting with the tools and begging for mercy.

In addition, I would like to thank Sandy Weisberg. From the beginning, she has been a big help in getting me through the administrative business that goes along

with an advanced degree. Plus, anytime I saw her in the hall she would always greet me with a smile and ask how I was doing. This helped me get through the day.

I would also like to thank my mom, Lea, for all of her love, patience, and understanding. Her constant encouragement and support throughout the years have given me the determination and strength to complete this difficult task.

Finally, I would like to thank Sue, my loving wife of only five months. I'm sure she did not realize what she was getting herself into by marrying a Ph.D. student during his last five months of graduate school. However, her love and dedication has provided me with an emotional support that I could not have done without. Having gone through this experience with her, I know that I can depend on her to be there whenever times get tough.

ABSTRACT

Signature_____

Advisor: Dorothy E. Setliff

USING MACHINE LEARNING TO DERIVE EFFICIENT COST AND PERFORMANCE ESTIMATES IN VLSI CAD DESIGNS

Donald S. Gelosh, Ph.D.

University of Pittsburgh

Area and delay estimates facilitate effective decision-making ability in high level synthesis. Current estimation techniques focus on modeling the layout result and fail to deliver timely or accurate estimates. This thesis presents a novel approach to deriving these area and delay estimates by modeling the actions and activities of the layout tool, rather than the layout result. This approach uses machine learning techniques to analyze the input-to-output relationships that result from applying the target layout tool to an input design description and producing a layout as an output. This thesis describes a solution architecture using these machine learning techniques that captures the relationships between general design features (e.g., topology, connectivity, common input, and common output) and layout concepts (e.g., relative

placement). This solution architecture has the following characteristics. First, a set of several training designs captures the general design features. The target layout tool is run on the training designs and produces a set of actual layouts. The general design features and relative placement concepts from the actual layouts makes up a training set. The formulation of this training set is important to adequately describe the set of general design features and the associated layout concepts. Second, a machine learning system analyzes the training set looking for relationships between the design features and layout concepts. This analysis produces a model of the operation of the layout tool. This model is in the form of a set of production rules. Third, this model is applied to real designs to formulate area and delay estimates. This approach is found to produce accurate area and delay estimates very quickly, even for designs with several thousand gates. Experimentation illustrates the identification of the general design features, the relative placement concepts, the formulation of the training set, the derivation of the tool model, and the application of this model to real world designs. In addition, two different layout tools were modeled to show the generality of this approach.

DESCRIPTORS

Area and Delay Estimates
Layout Tools
Modeling
CAD
Machine Learning
VLSI

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 INTRODUCTION

All design automation systems, independent of the domain, must produce a physical artifact with real physical characteristics. The earlier knowledge, or estimates, of the final physical characteristics are known and available for use in the design automation process, the more optimal the design process, as well as the design itself. Unfortunately, current VLSI CAD design automation systems fail to incorporate accurate estimates early on in the design process, namely, in high-level synthesis. The lack of accurate estimates at this level cripples the ability of the high-level synthesis system to produce optimal designs in a timely manner.

McFarland, Parker, and Camposano [1, 2]* define high-level synthesis as taking a description of a design's behavior along with cost and performance constraints and producing a structure that implements the behavior while satisfying the constraints. In order to find an acceptable structure, high-level synthesis systems give developers the ability to probe the design space of possible designs and to evaluate these designs. These evaluations are necessary to determine which designs are acceptable in terms of satisfying cost and performance constraints. (Cost and performance constraints set the upper and/or lower bounds on physical characteristics such as total chip area and critical path delay.) In order to determine the acceptability of a design, the high-level synthesis system must have the capability to measure each design's unique physical characteristics.

The most accurate measurements of a design's physical characteristics come from fully functional and fabricated chips. However, the layout and fabrication process is very time-consuming. In addition, any fabricated chips that do not meet the physical constraints set by the designer become costly, wasted material. To avoid wasting time and materials, estimates of the design's physical characteristics are used in place of the actual values. Only those designs whose estimates satisfy the physical constraints are fabricated and fully measured for compliance.

*Parenthetical references placed superior to the line of text refer to the bibliography.

Although it is possible to derive these estimates at different levels of the synthesis process, these estimates become more accurate as the design gets closer to being fabricated. The most accurate estimates for a target design are based on the layout mask that chip manufacturers use to actually build the chips. However, producing this layout mask is very time-consuming so it is not very effective to produce a layout for each design in the solution space. On the other hand, high-level synthesis systems that derive estimates at levels higher than the layout level to save time compromise their effectiveness because the estimates are not very accurate. Therefore, high-level synthesis systems can be ineffective in one of two extremes: they either use detailed information from actual layouts (time-consuming) or they use high-level estimates (inaccurate) to guide the synthesis process. These extremes define the end points of the estimator spectrum. Current high-level synthesis systems fall somewhere between these two extremes.

For example, the Fasolt system [3, 4, 5] automatically optimizes the scheduling and bus topology of a register transfer level design. Fasolt uses information taken from a layout model to perform this optimization. However, Fasolt uses a layout estimator and not an actual layout tool to produce the layout model. Knapp was more interested in quicker turn-around times for layout information than in high-quality layouts using slow layout tools. Fasolt's use of a layout estimator places this system near the extreme of producing actual layouts. (This and the other two systems addressed in this introduction are described in more detail in Section 2.1.)

The PLEST system [6, 7] estimates a design's layout area. This system is based on the standard cell design style. PLEST uses two estimators: one estimates wiring space requirements for the routing channels and the other estimates the number of feedthroughs. Even though this is an estimator that does not produce a layout, PLEST relies heavily on placement and wire routing techniques. PLEST uses a 1-D placement model and folds it to achieve the desired number of rows. PLEST then makes the area estimate based on this folded placement model. Kurdahi and Parker claim that PLEST is a fast system with an accuracy within 10% of the actual areas. The method, speed, and resultant accuracy places the PLEST system somewhere in the middle, away from both extremes. The PLEST system has accuracy approaching that of actual layouts, but it is fast and does not produce a layout.

The ADAM synthesis system [8, 9] predicts system-level area and delay without producing any layout information at all. The ADAM system uses a mathematical model to predict an area-delay trade-off curve for pipelined and nonpipelined data paths. This trade-off curve determines the lower bound (theoretically optimal) for a given data flow graph and a given module set. Jain, Parker, and Park admit their area and delay models are high-level, but the resultant curve serves to narrow the search space and arrive at an acceptable solution quickly. The high-level nature of this area and delay predictor places the ADAM system closer to the high-level estimator extreme.

Each of these systems resides in a different place in the area and delay estimator spectrum. The optimal location in this spectrum is where estimates are both accurate (based on actual layouts) and quickly obtained (no layouts produced). This thesis describes a method of quickly obtaining accurate area and delay estimates without producing a layout. This places this method near the middle of the estimator spectrum. In addition, this is a novel method because it uses machine learning techniques to model the input-to-output relationships that result from applying *layout tools*, rather than analyzing the *layout artifacts*. We have demonstrated that machine learning techniques make it possible to learn, for a given layout tool, what physical and graphical (size, topology, connectivity, etc.) features in a set of training designs affect the relative placement of nodes in the resultant layouts. This information (represented in the form of production rules) becomes a model that can be used in place of the layout tool to derive area and delay estimates. When a designer applies this model to a target design, the rules predict the relative placement of nodes based on what was learned from the training sets. The estimator uses this relative placement information to estimate area and critical path delay for the target design. We have shown that applying the model and obtaining area and delay estimates takes much less time than producing an actual layout. In addition, the area and delay estimates are accurate because they are based on information learned from actual layouts. According to a paper on layout estimation that Nourani and Papachristou presented at the 30th ACM/IEEE Design Automation Conference [26] in 1993, a difference of less than 10% is considered very good, but difficult to achieve. A difference of 20-30% is considered by the community [67] to be both acceptable and achievable. Therefore,

as a challenge to this research, we have established a goal of no more than 30% error in the area and delay estimates while trying to get most estimates within a 10% error.

The remainder of this thesis is organized as follows. Chapter 2 discusses background information on area and delay estimation techniques in current high-level synthesis systems, layout tools, and machine learning. Chapter 3 presents an overview of the tool modeling method and the general solution architecture. Chapter 4 describes the solution architecture in detail, as it was implemented for this research. Chapter 5 describes the set of experiments that helped to develop and validate this method and their results. Chapter 6 summarizes this thesis and discusses the contributions from this research, parts of the research that we could have done differently, and ends by suggesting future research.

2.0 BACKGROUND

The area and delay estimation method developed by this research uses machine learning techniques to model the input-to-output relationships that result from using layout tools. Using a model instead of an actual layout tool, VLSI chip designers can obtain layout information much faster than producing an actual layout. The designers can also quickly obtain accurate area and delay estimates from this layout information and use these estimates to guide design decisions in high-level synthesis. Therefore, this research is based on prior work in the fields of area and delay estimation in high-level synthesis, layout tools, and machine learning. Section 2.1 discusses prior work in high-level synthesis focusing on claims, successes, and limitations of major thrusts in area and delay estimation. Section 2.2 discusses specifics on layout tools. Section 2.3 describes machine learning in general and discusses prior work involving the machine learning system used in this research.

2.1 Area and Delay Estimation in High-Level Synthesis

McFarland, Parker, and Camposano [1, 2] define high-level synthesis as taking a description of a design's behavior along with cost and performance constraints and producing a structure that implements the behavior while satisfying the constraints. In order to do this, a high-level synthesis system must have the capability to evaluate the cost and performance of designs in the solution space. (For this research, cost is evaluated in terms of area and performance is evaluated in terms of critical path delay.) To save valuable design time in exploring a large solution space, the high-level synthesis system uses estimates. This section discusses several area and delay estimators and how they can be used in high-level synthesis systems.

The Fasolt system [3, 4, 5] is a data-path optimizer. It uses a layout estimator to produce a layout model. This layout estimator is a compromise between a floorplanner and a macro cell placement tool. According to Knapp, "It generates a macrocell-oriented placement; performs module orientation, aspect ratio assignment,

and pin assignment; constructs a channel graph; and estimates channel widths by performing loose global routing." Knapp considered using a slower, high-quality layout tool, but decided to use a primitive layout estimator instead. He did this to save time in exploring the design space.

The layout estimator can use either a min-cut placement [10, 11, 12] or simulated annealing [13, 14, 15] placement strategy, selected by the user. A Steiner tree [16] approximation is used for multi-pin nets in global routing. The channel widths are estimated by summing all wires in the channel. This is based on a pessimistic assumption that all wires extend along the entire length of the channel. All of this information makes up the layout model. The layout model provides estimations of low-level physical properties so Fasolt can evaluate the area and delay of a particular data-path optimization.

However, even though the layout estimator does not produce a layout, it still takes a long time to perform placement and routing. This makes the data-path optimization a slow process. As a result, Fasolt is limited to a macrocell layout style so the layout estimator only needs to handle a small set of large modules. This means this type of approach will not work well in a full custom gate or standard cell environment. In addition, Knapp did not compare results from the layout estimator to results from a real layout tool. Knapp did state this would be part of planned future research.

In the ADAM [8, 9] synthesis system, a prediction tool that focuses on the sums of areas and the sums of delays for individual components is used to predict a lower bound area-delay trade off curve. A lower bound curve means that all design points must lie on or above the curve. All design points on the curve represent optimal designs. Jain, Parker, and Park admit this is a high-level predictor and it only considers whole components when predicting the total area and delay. While it serves to narrow the search space and help the system to arrive at an acceptable solution, it does not come close to the accuracy or design quality obtained from a full-custom layout. In addition, while it considers functional units and controllers in the area and delay predictions, it does not consider registers, multiplexers, buses, or wiring. Also, because this system only produces a lower bound curve, the only way to compare it to other layout tools is to plot a similar curve. While Jain et. al.

plotted curves for other layout tools and compared them to curves from the ADAM system, no percentage differences were calculated.

The PLEST system [6, 7] only estimates the area requirements of VLSI designs. This system is based on standard cells. PLEST uses two estimators: one estimates wiring space requirements for the routing channels and the other estimates the number of feedthroughs. PLEST uses a 1-D placement model [17, 18] and folds it to achieve the desired number of rows. A 1-D placement is easier to model than a 2-D placement and Kurdahi and Parker claim this allows them to analyze the design better and produce confident area estimates. Kurdahi and Parker also claim PLEST is a fast system with an accuracy within 10% of the actual areas.

In the TELE 2.0 system [19, 20], Ramachandran and Kurdahi use a combination of constructive (partial slicing tree) and analytical (Rent's rule [21]) approaches to estimate wire-length. A slicing tree is built from the netlist information. However, the design is sliced only to a small number of levels. They call the leaves of the tree the *leaf clusters.* Analytical area and shape function estimators [6, 7] are applied to the leaf clusters to obtain area estimates. By traversing the tree from the leaf clusters to the root, TELE 2.0 can find the shape function for the root node. Once this is done, TELE 2.0 can approximate the location of the leaf clusters. This location information determines the layout net lengths between leaf clusters. The length of the wires inside the leaf clusters is estimated by using Rent's rule. TELE 2.0 uses these wire-length estimates along with individual cell delays to provide a delay time for the worst case delay path in the design based on the circuit topology. Ramachandran and Kurdahi claim TELE 2.0 is a fast and reasonably accurate tool (7% or better), but only for small designs (less than 1800 cells). TELE 2.0 is also limited to a standard cell environment.

Ramachandran et. al. [22, 23] also have a tool that models layout area and delay for structural register-transfer level designs. It uses constructive techniques to incorporate floorplanning information into the area estimation model. To its advantage, this system models the required area for the data-path, controller, macrocells, and memories. It can also directly estimate the total chip area. This system calculates delay in three parts: intrinsic cell delay, fanout delay, and wire length delay. The intrinsic cell delay and fanout delay comes from a cell library and the netlist. The

wire delay is based on constant width wires and a lumped RC model [24, 25] where the capacitance and the resistance are directly proportional to the length of the wire. The approximate layout topology generated by the area model determines the wire length estimate.

While the authors report worst case errors of 13% for area estimates and 10% for delay estimates, they state that they cannot make a general statement about the efficiency of this system because it has only been tested with a limited range of designs containing less than 2500 cells.

Nourani and Papachristou developed an algorithm [26] that estimates layouts for register-transfer level datapaths using standard cells. It uses a non-probabilistic analytical formula in a constructive algorithm. The estimation algorithm executes in two phases. Phase 1 places all cells in one row and Phase 2 folds them into the user-specified aspect ratio. These are the classical placement phases for estimators, but Nourani and Papachristou claim theirs is different. They do not use any probabilistic methods to estimate wire lengths. Instead they use information about the input and outputs ports of the cells from the cell library. They also use a flexible layout model for the components and do not need to consider feedthroughs. This requires less computations and saves time. Another difference is that they can estimate layouts based on either area minimization or delay minimization, while using the same model.

Nourani and Papachristou claim their area estimates are within 12% of actual layouts, but they can only show this for small designs (less than 2100 cells). They also claim that it can be extended to handle full-custom cells, but they must define these cells in terms of standard cells.

None of the systems described above considered modeling the relationships that result from using layout tools instead of analyzing the target designs or estimating layouts to gain a time and resources advantage. Our research shows that modeling these relationships and using the model in place of the layout tool is a reasonable and efficient alternative in estimating the area and delay of target designs.

2.2 Layout Tools

Layout tools perform the last step in the VLSI physical design process [2, 23, 27]. Layout tools transform a low level circuit description into a mask that chip manufacturers use to guide the chip fabrication process. The circuit description may be a netlist of primitive gates or standard cells or even higher level macro cells. The layout tool analyzes this netlist and performs two tasks: placement and routing. The layout tool places the gates or cells in such a manner that some cost function is satisfied. This cost function could be a function of the total area of the layout [28, 29, 30], or a function of the critical path delay [27, 31, 32]. Once the gates or cells are placed, the layout tool routes the appropriate wires between the gates or cells. This routing is also performed to satisfy some cost function.

However, many types of layout problems such as global routing [30, 33] and minimizing the channel width in Manhattan routing [34, 35], are known to be NP-complete [36]. This makes it very difficult to find the exact optimal solution. Therefore, we require heuristic techniques to search the solution space in an attempt to find an acceptable solution. In some cases, the solution space is quite large and even with a heuristic search, it takes a lot of computational time and effort to find an acceptable solution.

2.2.1 Simulated Annealing

One such heuristic technique is simulated annealing [13, 14, 15]. This technique incorporates a probabilistic hill-climbing search method [37]. The simulated annealing algorithm is shown in Figure 1. The first step is to build a random initial configuration (in this case, an initial layout). This becomes the current configuration. A new configuration is built by randomly permuting the current configuration. The new configuration is evaluated to see if it is the best at satisfying the cost function so far. If it is the best, it is saved. Then the algorithm determines if the new configuration should replace the current configuration. It does this through an acceptance function. The acceptance function algorithm is shown in Figure 2. The decision to accept the new configuration is based on the cost of the current configuration,

the cost of the new configuration, and the current temperature. At first, the temperature is a large value. When the temperature is high, the new configuration is more likely to be accepted, even if it has a worse cost than the current configuration (hill-climbing). As the simulated annealing process continues, the temperature is cooled down according to a cooling schedule (the update function line in Figure 1). As the temperature cools, the likelihood of accepting new configurations decreases. Therefore, the algorithm starts to behave like a greedy algorithm in the final stages.

```
Simulated-Annealing(T0)
{
  Build initial configuration cur_conf;
  T = T0;
  sav_conf = cur_conf;
  while (stopping criterion is not satisfied) {
    while (inner-loop criterion is not satisfied) {
      new_conf = permute(cur_conf);
      if (cost(new_conf) < cost(sav_conf)) {
        sav_conf = new_conf;
      }
      if (Accept(cost(cur_conf), cost(new_conf), T) == TRUE) {
        cur_conf = new_conf;
      }
    }
    T = update(T);
  }
} /* end Simulated-Annealing */
```

Figure 1 Simulated Annealing Algorithm

While this is a simple straightforward algorithm, the number of permutations required to produce quality results grows with the size of the layout. In addition, evaluating the cost function for large layouts can be very time-consuming. While they can produce quality results, layout tools based on simulated annealing take a long time. For example, ArtistII [27, 28, 29, 38], uses a variant of simulated annealing. As described in Section 3.3.2.1, ArtistII required a couple of weeks of CPU run time to process a design with only about 3000 gates. This time requirement makes it difficult to use area and delay values based on actual layouts in high level synthesis. However, modeling the relationships resulting from using the layout tool and using this model

```
Accept(cur_cost, new_cost, T)
{
  delta_cost = cur_cost - new_cost;
  y = exp(-delta_cost/T);
  r = random(0,1);
  if (r < y) {
    return TRUE;
  } else {
    return FALSE;
  }
} /* end Accept */
```

Figure 2 Accept Function Algorithm

in place of the tool to produce layout information saves time. The next two sections, Section 2.2.2 and Section 2.2.3, describe the two target layout tools we used in this research.

2.2.2 ArtistII

ArtistII is a second-generation CMOS layout tool developed at the Pennsylvania State University [27, 28, 29, 38]. A detailed description of the operation of ArtistII is given in the appendix. This layout tool uses a variant of simulated annealing to automatically map a given design into a layout while optimizing the layout area. ArtistII analyzes a transistor-level net-list description of the target design [39], optional gate-clustering information [40, 41], and a set of user specified parameters [42], including the number of trial placements to evaluate and the number of rows to use in the layout. The output is an object-based description of the smallest layout found during the user specified number of trial placements. ArtistII also includes a translator that translates this object-based description into a Magic [43, 44] format layout file and a grouping description file. The grouping file indicates the row and column location of each primitive gate in the layout.

ArtistII was developed for use on small to medium size designs using no more than a few thousand primitive gates. It works best on small designs (fewer than 100 gates), but it is capable of handling larger designs as well. The larger designs

require longer running times for good quality results. Table 1 shows the run times (in CPU seconds) for a set of designs used in this research. These run times were obtained by running ArtistII on a Sun SPARCstation 2 with 593 Megabytes of swap space. The first column lists the names of these designs. The names also indicate the function of the design (i.e., *mult16* indicates a 16-bit multiplier design). The second column shows the number of primitive gates in the design. The third column shows the number of transistors. The last column shows the required run time for ArtistII. For example, the design *mult8* has 711 gates made up of 2574 transistors. It took ArtistII 47696 CPU seconds (a little over 13 CPU hours) to produce a layout. All of the run times, except for the last three designs, are based on using 10,000 iterations. (ArtistII considers an evaluation of one trial placement as an iteration. Therefore, one iteration would equate to one pass through the outer loop in the simulated annealing algorithm shown in Figure 1.) The last three designs, *mult16*, *big*, and *realbig* use only 100 iterations. This is because the run time of ArtistII using 10,000 iterations for these designs made it impossible to obtain a layout in the local computing environment.

This table shows that the run time for ArtistII can become prohibitively long, even for medium size designs having only several hundred gates. It would be impractical to expect a designer to wait this long for results, especially if there are several designs in the solution space requiring evaluation. It would be better to use a model of ArtistII to quickly obtain layout information for the designs in the design space, and then use ArtistII to produce a quality layout for the selected design.

2.2.3 TimberWolf

The TimberWolf layout tool was developed as a timing driven placement and global routing package [30, 45, 46]. It is capable of handling any of the row based designs including standard cells, gate arrays, and sea-of-gates designs. TimberWolf is also applicable to floorplanning problems and may be used to completely place and globally route mixed macro/standard cell designs.

Placement of standard cells and global routing in TimberWolf takes place over three distinct stages. In the first stage, TimberWolf uses simulated annealing to place

Table 1 Execution Times for ArtistII

Design Name	Number of Gates	Number of Transistors	Execution Time (CPU sec)
add4	43	146	840
add8	95	330	2278
add12	147	514	5385
add16	199	698	8278
add24	303	1066	11886
add32	407	1434	18735
abs16	208	638	6676
inc4	23	68	405
inc8	51	156	980
inc16	107	332	3408
sub4	48	160	828
sub8	104	352	2374
sub16	216	736	7788
addsub4	64	224	1398
addsub8	132	464	4896
addsub16	268	944	10224
mult2	40	112	653
mult4	163	550	4106
mult6	389	1378	25213
mult8	711	2574	47696
mult16	2959	11038	28746
big	6264	23388	65493
realbig	12528	46776	359289

cells while minimizing the total interconnect cost. In the second stage, TimberWolf inserts feed-through cells as required while continuing to minimize total interconnect cost through simulated annealing. TimberWolf also uses uncommitted feed-throughs from the standard cells in place of additional feed-throughs whenever possible. In the third stage, TimberWolf makes local changes to the placement if these changes lead to a reduction in the number of wiring tracks.

Sechen claims that, when compared to other automatic [47, 48] and manual layout methods, TimberWolf yielded an area savings from 15 to 75 percent. He also claims the global router reduced the number of wiring tracks by an additional 7 to 16 percent when compared to the total interconnect length minimization alone. In addition, he claims the third stage (local placement refinement) reduced the number of wiring

tracks another 8 percent.

Now that we have described what we are trying to model, the next section describes various machine learning systems that we could use. The next section first describes what machine learning is all about and then describes three different implementations of machine learning. The section finishes by describing the machine learning system we used in this research.

2.3 Machine Learning

A machine learning system is a computer program that makes decisions based on the accumulated experience contained in successfully solved cases [49, 50, 51, 52]. This experience is represented as a set of patterns of data with correct classifications (based on the solution) for each data pattern. Thus, learning becomes a classification problem. The learning system decides which classification to make for a given data pattern. This classification problem is called a *prediction problem* in statistics and *concept learning* in machine learning. Each data pattern has a particular classification or concept associated with it. The task of the learning system is to learn what data patterns produce which concepts.

The data patterns represent observations about certain features in the target domain. Each feature is described by an attribute (color, size, shape, etc.), which can take on a value (red, big, square, etc.). Of course, each feature has a range of admissible values. These values may be discrete, as in color, or continuous, as in size. Each set of feature values and their associated concept is called a *training instance.* A complete set of training instances (accumulated experience) is called a *training set.* All training instances in the training set must have the same set of features, but not necessarily the same set of values.

The learning system analyzes the training set and produces a classifier. This classifier makes classification decisions based on the learning system's acquired knowledge of the solved cases. The classifier takes a set of feature values as input and produces a concept as output.

Producing a classifier would be an easy task if the learning system had an unlimited number of training instances from which to learn and if the domain knowledge contained in these training instances was complete. However, in the real world this is never the case. The learning system has only a limited number of training instances and the domain knowledge is lacking in some respect. Therefore, there are two basic problems in machine learning. One is trying to generalize decisions based on only a sample of specific training instances. The other is trying to make the correct decisions based on incomplete knowledge.

If the training set could include all possible training instances with the correct concepts, the classifier could be implemented as a simple lookup table. The classification process would consist of matching the input set of features with the same set from the lookup table and using the associated concept as the output. This would always produce correct results. However, unless the number of features and concepts for a particular domain is very small, this type of pattern matching becomes a difficult problem. It becomes impossible if any of the values are continuous. Therefore, because training sets can reflect only a sample of all possible training instances, the problem becomes one of trying to generalize decisions based on a limited number of specific training instances. This makes the classifier more complicated and introduces errors.

The error rate of a learning system is the most commonly used measure of success or failure [50, 52]. When the classifier tries to classify an unknown case, it will either make the correct choice or it will make an incorrect choice. The error rate is defined as the ratio of the number of incorrect choices to the total number of unknown cases. A true error rate could be calculated if the there were an unlimited number of cases, all with the correct concept already known. However, this is not practical, so an apparent error rate is derived. The apparent error rate of a classifier is the error rate of the classifier on the training set [49, 52]. This makes sense because as the number of instances in the training set approaches infinity, the apparent error rate will eventually become the true error rate. In this research, we determine the error rate of the learning system in a slightly different way. Instead of measuring how a classifier performs on the training set, we measure the effects of its performance on real cases. That is, we measure how well our area and delay estimator performs,

given predicted layout information from the learning system. This is explained in more detail in Section 5.2.

Another problem arises when the acquired knowledge about a domain is incomplete and the classifier cannot make a decision with certainty [49, 52]. For example, the same set of feature values may show up in different training instances with different observed concepts. In this case, the classifier must determine which concept is most likely to occur for that particular set of feature values when encountered in the unclassified cases. This introduces the need for expressing confidence in the classifier's decisions. This problem is solved by using certainty factors.

Certainty factors combine degrees of belief and disbelief about a decision's truth into a single number [49, 53, 54]. This number allows comparisons of competing decisions. When the learning system produces a decision, it assigns the decision a certainty factor based on acquired knowledge in the training set. Then, in the classifier, if different decisions produce different concepts for the same set of feature values, certainty factors help to select a concept. For instance, if a decision indicating one concept for a given data pattern has a high certainty factor and another decision indicating another concept for the same data pattern has a low certainty factor, only the decision with the high certainty factor is used. This is known as the *single best* decision scheme [53, 54]. The decision with the single best certainty factor is used.

Certainty factors may also be used in a *weighted voting* scheme [53, 54]. In this scheme, the certainty factors of all decisions indicating the same concept for a particular data pattern are combined into a single certainty factor. This is done for all decisions bearing on that data pattern. The concept with the highest combined certainty factor is selected as the concept for the data pattern. (This is the method we use in this research. This is explained in more detail in Section 3.3.2.2.)

Now that we have presented a general overview of machine learning, we can look at specific implementations. There are three major techniques for implementing machine learning classifiers. These techniques are linear discriminants, neural nets, and production rules. The following sections describe these techniques.

2.3.1 Linear Discriminants

Linear discriminants are the most common form of classifier [50, 52]. In these systems, a linear combination of the feature values discriminates the classes. If a domain has d features, then a $d-1$ dimensional hyperplane separates the classes. The general form is given in Equation 2-1, where $(e_1, e_2, ...e_d)$ is the set of feature values, d is the number of features, and w_i are constant weights learned from the acquired knowledge (training set) for the domain. Equation 2-1 implements a weighted sum of the feature values.

$$sum = w_1e_1 + w_2e_2... + w_de_d - w_o \tag{2-1}$$

The learning system produces a unique set of weights for each class. The classifier contains these sets of weights and computes this sum for each class on every unknown case presented. The class whose weights produce the highest sum becomes the selected class for that case.

The kinds of problems best solved by linear discriminants are ones in which the classes are normally distributed [50, 52]. A linear discriminant based learning system was used in Samuel's checkers program [55]. Basically, this program learned to play checkers by playing several games. When good moves occurred, features that were heavily weighted in the classifier got additional weight. When bad moves occurred, they lost some of their weight. The classes describing the different moves were assumed to be normally distributed. This was a simple problem and only required about a day of computation time. (Of course, this computation time is measured in 1959 time when computers were much slower.)

2.3.2 Neural Nets

Neural nets are another type of classifier. A neural net is a parallel distributed information processing structure in the form of a directed graph [52, 56, 57]. The nodes of the graph are the processing elements. The edges of the graph are unidirectional connections between processing elements. A processing element can have any number of input connections. A processing element can also have any number of output

connections, but each output must have the same value. (The processing element really has only one output. This output is fanned out as needed.) Each processing element has local memory and possesses a transfer function. This transfer function uses information from the local memory and the inputs to produce new information for the local memory and the output. Therefore, the transfer function implements the learning process. Learning takes place in a neural net as the local memory is modified each time a new training instance is presented. The training instances are presented sequentially and the order does not matter.

The classic implementation of a processing element is a device called a perceptron [50, 52, 56, 57]. Figure 3 shows a typical perceptron. A perceptron decides if an input pattern belongs to one of two classes. In this regard, a perceptron is the equivalent of a linear discriminant. Just as in Equation 2-1, a perceptron has a fixed number of inputs that correspond to the number of features in the domain. The inputs are weighted and summed to produce an output. The weights are stored in local memory and the transfer function performs the sum. If this sum is greater than 0, the output of the perceptron is assigned a value of 1, not the summed value. If the sum is less than or equal to 0, the output of the perceptron is assigned a value of 0. The output of a perceptron indicates to which class the set of inputs belongs. If the output of the perceptron is 1, the input pattern belongs to class 1. If the output is 0, the input pattern belongs to class 0.

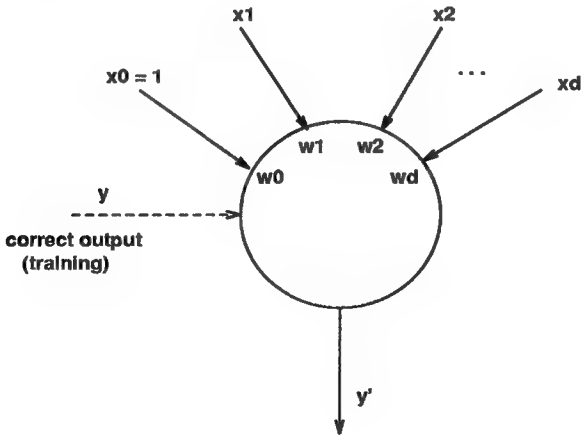


Figure 3 A Perceptron

During the learning process, there is another input to the perceptron. This input indicates the correct output value for the input pattern, based on the observed cases in the training set. In the learning mode, the transfer function in the perceptron computes the weighted sum of the input pattern and compares the resultant output to the correct output value. If the perceptron's output is the same as the correct output, the weights are not changed. If the perceptron's output and the correct output are different, the transfer function adjusts the weights and stores them back in local memory. Thus, the next time the perceptron is presented with the same input pattern, it will produce an output closer to the correct one. The inventor of the perceptron, Frank Rosenblatt [58], proved that given linearly separable classes, a finite number of training instances will develop a set of weights that will always produce the correct outputs. This is known as the perceptron convergence theorem. Convergence to the correct set of weights will eventually occur even if the initial set of weights are randomly chosen. However, it is not known a priori how many training instances are needed to converge or how long it will take to converge.

Neural nets, unlike linear discriminants, make no assumption about the population distribution. The classes do not have to be normally distributed to get good results. Neural nets are well suited for such problems as sensor processing, control applications, and data analysis [56, 57]. For example, one sensor processing application is character recognition. The user forms a character on a digitizer tablet and the neural net recognizes the character and uses it as appropriate. Of course, the training set would consist of many such characters.

One drawback to both linear discriminants and neural nets is the decision rules are hidden. In both systems, the weights determine the classification for a given set of feature-values. These weights are based on what was learned from the training set. But these weights do not directly indicate what relationships exist between the features and the concepts according to accumulated experience in the training set. There is no way to directly check the logical correctness of the decisions based on the weights. Therefore, the classifications from linear discriminants and neural nets must be accepted on faith if there is no known correct answer to compare them against. The decision rules could be inferred by an involved mathematical computation, given the user is so inclined and has a mathematics background.

A better solution is to implement a learning system that produces rules directly, and makes the rules compatible with human reasoning in a human readable format. This way, the user can easily check the rules to determine if the rules make sense for the domain. If the learning system produces rules that do not make sense for the domain at first glance, the rules can be investigated further. This investigation will either show that the rules are not correct, or it will show that the rules have uncovered something not well known about the domain. In the first case, we can disregard the rules and either adjust the learning process or modify the training set to produce rules that do make sense. In the second case, we can add the new information to our accumulated knowledge of the domain and use the rules with confidence.

Due to the novel use of machine learning techniques in the VLSI CAD domain in this research, we found it necessary to check the rules directly. Therefore, we used a machine learning system that produces human readable rules. Section 2.3.3 discusses rule-based learning systems in general and Section 2.3.3.1 discusses the learning system used in this research.

2.3.3 Rule-Based Machine Learning

In rule-based learning systems, decision rules have the general form of an "IF-THEN" statement [49, 51, 52]. IF X is true, THEN classify as class A. The simplest form of the "IF" part is a conjunction (AND) of propositions: IF X is true AND Y is false, THEN classify as class B. However, more than one rule in conjunctive form may be required to properly handle a class. The required rules may be put together in a disjunctive (OR) set. This format for combining rules into sets is called the *disjunctive normal* form. Learning systems that produce rules in this form make it easy for humans to understand the rules and to have confidence in the learning process.

There are two basic techniques for implementing rule-based learning systems: decision trees [57, 59] and production rules [52, 60]. Decision trees are directed graphs where each node represents a single decision and the edges indicate where to go based on the decision. Leaf nodes indicate a class decision. All paths in a decision tree end up at a leaf node. The leaf node indicates which class is selected corresponding to a

conjunction of decisions made along the path. If there are multiple paths for a given class, the paths represent disjunctions. In addition, all paths in a decision tree are mutually exclusive. One and only one path will eventually lead to a classification for a given set of feature-values. This also means that only one classification can be selected for a given set of feature-values. Therefore, there is no need for certainty factors or rule weighting schemes.

The goal of a decision tree based learning system is to learn the structure of the tree. This process is known as rule induction. The tree is induced from the training set. A starting feature is selected for the root node. The root node is split into disjoint sets according to distinct values for the feature. This process is repeated for subsequent nodes and features. Leaf nodes occur when all members of the sample belong to the same class so splitting is no longer necessary. This class becomes the selected class for the path that lead to the leaf node.

Using production rule techniques gives the learning system more power. This is because production rules do not have to be mutually exclusive like paths in a decision tree. For any given input case, more than one rule may fire for the same class, or two or more rules may fire for different classes. (A rule is said to "fire" when the IF part is satisfied by the input data pattern.) If two or more rules fire indicating different classes, we need an ordering scheme or certainty factors to help select the most appropriate class.

Building a learning system using production rules is quite complex. This is because it is difficult to predict the interaction of rules that are not mutually exclusive, even on the training set. However, if an exhaustive search of all possible rule sets in disjunctive normal form is implemented, the power of the production rule system becomes evident. All possible rules would be generated from the most general to the most specific. These rules would cover all of the possible relationships from features to concepts. Of course for nontrivial domains, an exhaustive search is impractical, so we need to use heuristic search techniques. One heuristic technique is to use a beam search [52, 53, 54]. In an exhaustive search [52, 59], logical expressions are generated and discarded if they are no better than the best expression so far. All possible expressions are generated and compared and the best one is kept. In a beam search, the most promising expressions are stored. (An integer number called a beam

width indicates how many expressions to keep in storage.) Combinations of these expressions are used to generate longer expressions. The most promising expressions generate even longer expressions and so on, until some previously defined expression length is reached. Then the best rule in this final set is put into the classifier.

Section 2.3.3.1 describes the production rule learning system used in this research. This learning system proved to be very useful and quite powerful for our application.

2.3.3.1 Rule Learner. The machine learning system used in this research is called Rule Learner [53, 54] and was developed by the Department of Computer Science at the University of Pittsburgh. It is a descendent of the Meta-DENDRAL learning system [51, 60]. The Meta-DENDRAL system was successful in finding new cleavage rules for use in mass spectrometry. Mass spectrometry is a method for identifying the chemical constitution of a substance by means of the separation of gaseous ions according to their differing mass and charge. Cleavage rules are used to predict which bonds in the substance will be broken [50, 60].

The Rule Learner system is a knowledge-based production rule system. Rule Learner learns a set of rules based on a training set through heuristic search techniques. The knowledge is made up of the kinds of rules that are plausible plus the training set itself. There is also a set of user-specified stopping conditions (biases) that indicate when a set of rules adequately describes the training set. To date, Rule Learner has been successfully used in High-Energy Physics experiments [54]. It was used to diagnose errors in the beam line of a particle accelerator, design diagnosable beam lines, and design the experiments themselves. The high-energy physics events formed the examples for learning and the features used to describe the examples were physics quantities. The Rule Learner system has also been used to predict the carcinogenicity of chemicals and to diagnose errors in the local loops of telephone networks.

However, the Rule Learner system does have some limitations. It is unable to learn rules that relate one feature to another. Therefore, relational features must be explicitly declared in the training set. In addition, the running time grows exponentially as a function of the number of conjuncts, attributes, and values. We

can calculate the order of this running time for Rule Learner's search algorithm [53] by assuming there are a attributes, each having a maximum of v nominal values, n training instances, and a maximum of k conjuncts per rule. Then the search tree has a depth k and a maximum branching factor of av. If an exhaustive depth-first search were used, the run time would be $O((av)^k n)$. However, the Rule Learner version we used in this research implements a beam search. A beam width of b reduces the run time to $O(kbav(n+\log bav))$.

The Rule Learner system has proven to be very appropriate for this research because it can intelligently search through a large set of data and find rules that explain the data. This is well suited to analyzing training designs and their resultant layouts so a model of the relationships from the layout tool can be represented by the rules learned. Rule Learner also produces rules in a clear and human-readable format so we can understand them, check them to ensure they make sense, and use them with confidence. Implementation of Rule Learner in this research is described in more detail in Section 3.3.2.

3.0 RESEARCH APPROACH

The goal of this research is to find a method for quickly obtaining high quality area and delay estimates so we can increase the effectiveness of high-level synthesis systems. Section 2.1 describes several area and delay estimators that analyze designs and/or model layouts. None of these estimators consider modeling the input-to-output relationships that result from using layout tools instead of analyzing the design or the layout artifact. The premise of this research is a belief that we can model these relationships from layout tools and use these models in place of the tools to quickly obtain accurate layout information. High-level synthesis systems can effectively use this layout information to obtain area and delay estimates.

This approach has a significant time advantage because it does not produce an actual layout. It does not rely on time-consuming placement and routing techniques. This approach also has an accuracy advantage because it does produce and use low-level layout information based on actual layouts. It does not rely on inaccurate and high-level models of area and delay.

Having decided on a method that models layout tools, we considered the following alternative approaches:

- Use layout tool in "brain-damaged" mode.
- Use library of designs and corresponding layouts.
- Perform input-to-output analysis of layout tool.

Each of these approaches is described in Section 3.1. This section describes how we carefully considered each approach before choosing one for this research. We selected the input-to-output analysis approach because it yields better and faster results than the other two. This approach also allows for greater flexibility because it can be applied across various design architectures and on different layout tools. Following the section describing the alternative approaches, Section 3.2 presents an

high-level overview and Section 3.3 presents the solution architecture for how we used machine learning techniques to model layout tools.

3.1 Alternative Approaches to Modeling Layout Tools

One approach to modeling layout tools is to use a "brain-damaged" version of the target layout tool. The principle here is to trade very accurate results for quickly obtained results. The target layout tool is scaled back so it produces a less than optimal layout in a shorter period of time. For example, a layout tool that uses a user-defined number of iterations in its algorithm may be scaled back by decreasing the number of iterations. Table 2 shows height and width measurements for a sample set of designs we processed through ArtistII [27, 28, 29, 38] using a different number of iterations. (Because ArtistII uses a cost function based on minimizing area, comparing delay values is meaningless.) The height and width measurements come from the Magic [43, 44] layout mask produced by ArtistII.

The first column shows the name of the design and the number of gates. The second column shows the area dimension (height or width) being evaluated. The third column shows the height and width values (in lambda units, 1 lambda = 2 microns) for only 100 iterations of ArtistII. Using only 100 iterations of ArtistII is considered a "brain-damaged" mode. The fourth column shows the percent difference between these values using 100 iterations and the values we obtained by using 10,000 iterations of ArtistII (col. 7). (We used 10,000 iterations as a reference here because we modeled ArtistII at 10,000 iterations.) The fifth column shows the height and width values for 1000 iterations. Using only 1000 is considered a "brain-damaged" mode, but it is a little smarter than using only 100 iterations. The sixth column shows the percent difference between values using 1000 iterations and values using 10,000 iterations. The seventh column shows the reference values using 10,000 iterations of ArtistII. Because 10,000 iterations is the reference number of iterations it is not considered a "brain-damaged" mode.

For example, one of the designs is a 16-bit incrementer named *inc16*. It has 107 gates. The reference values at 10,000 iterations are 1115 lambda for height and 520 lambda for width. ArtistII required 3408 CPU seconds to produce a layout using

Table 2 "Brain-Damaged" Results from ArtistII

Design Name (# gates)	Dim.	100 Iterations	Percent Error	1000 Iterations	Percent Error	10000 Iterations
add4	ht	555	20.92%	499	8.71%	459
(43)	wd	422	39.27%	359	18.48%	303
add32	ht	3179	6.71%	3259	9.40%	2979
(407)	wd	1640	9.33%	1563	4.20%	1500
abs8	ht	1074	8.81%	1083	9.73%	987
(104)	wd	695	33.65%	527	1.35%	520
dec16	ht	963	39.36%	723	4.63%	691
(107)	wd	1122	37.84%	891	9.46%	814
inc16	ht	1339	20.09%	1243	11.48%	1115
(107)	wd	856	64.61%	625	20.19%	520
mult8	ht	3971	9.48%	3859	6.40%	3627
(711)	wd	3663	11.98%	2963	-9.42%	3271

10,000 iterations. Using only 100 iterations which took ArtistII only 62 CPU seconds, we obtained a height of 1339 lambda and width of 856 lambda. These differ from the reference values by 20.09% for height and 64.61% for width. Using 1000 iterations which took ArtistII a little longer (353 CPU secs.), we obtained 1243 lambda for height and 625 lambda for width. These differ from the reference values by 11.48% for height and 20.19% for width. This shows that as the number of iterations is increased, the percent difference decreases. It also shows that as the number of iterations increases, so does the time required for ArtistII to produce a layout.

The results we obtained from using a "brain-damaged" mode of ArtistII are not very good, but may be acceptable as rough estimations. The biggest drawback to this approach is it still requires producing a layout. Therefore, very large designs still require a substantial amount of time and memory, even if the number of iterations is decreased. In addition, as we can see in Table 2, the accuracy decreases as we decrease the number of iterations.

Another approach is to use a library of designs and layouts. The target layout tool could process a set of typical designs to generate a set of corresponding layouts. The design descriptions and their corresponding layouts would be stored in the library. We would process a new design by matching it up with one of the designs in the library

and using information from the corresponding layout to estimate area and delay. One drawback to this approach is the amount of memory required for a library containing a sufficient set of designs and layouts. Another drawback is the time and processing required to perform the extensive search through the library while trying to match up designs, either in total or in part.

The last approach (the one we selected for this research) is to determine what input-to-output relationships result from using the layout tool. This requires using the target layout tool to process test designs into actual layouts, and then analyzing both the input test designs and the resultant layouts to determine what relationships occur between them. These relationships form the model. When applied to an actual design, this model will produce sufficient information describing a predicted layout from which we can obtain area and delay estimates. Applying the model and estimating area and delay in this way is orders of magnitude faster than just producing an actual layout, even for large designs.

However, while this approach is general and can be abstracted to other layout tools, a set of input test designs that sufficiently represent the expected designs and a data analysis tool are required. For this research, the expected designs are simple microprocessor aritmetic logic unit (ALU) components using single cycle clocks with one instruction per cycle. Therefore, the suite of test designs (described in Section 5.2) is made up of these components.

We considered three approaches for the data analysis tool. One approach was to build a custom data analysis tool from scratch. While this would serve the purpose, it would be too specific to one tool to be abstracted to other tools. A separate analysis tool would have to be built for each layout tool being modeled. Another approach was to use an existing database analysis tool to discover trends in the layouts based on their corresponding designs. The drawback to this option was performing the extensive but necessary modifications to customize this analysis tool so it could function on the target layout tool. This also made this approach too specific. A third approach was to use a general purpose machine learning system. As it turns out, we can readily customize this type of system to function on a wide range of layout tools.

Having selected the machine learning approach for our data analysis tool, we considered three types of learning system implementations: linear discriminants, neural nets, and production rule based systems [49, 50, 51, 52]. As explained in Section 2.3, we could use any of these systems to learn from a training set how design features map to layout concepts. However, in the case of linear discriminants and neural nets, the underlying rules would be hidden. This makes it very difficult, if not impossible, to verify the rules make sense and can be used with confidence. For this research, we found it necessary to verify the rules when learning from a limited number of designs. For example, all of the training designs in a single training set may have a certain combination of features that appears to affect their resultant layouts, when in fact it is just a coincidence. As a result, the learning system would learn this coincidence as fact and produce a rule that modeled it. However, this rule would be valid only for the training designs and not hold for others. In other words, the rule would be too specific and not general enough to apply to all designs. If this situation occurred while using linear discriminants or neural nets, the apparent error rate would be low while the true error rate would be high (see Section 2.3). It would be very difficult to discover why. If we were using a production rule system, we could judge the rule directly to determine why the error rates were different.

The approach using the production rule based learning system proved to be the best solution for other reasons. Unlike linear discriminants or neural nets, production rule systems allow the same set of features to be mapped to more than one layout concept [50, 52]. This is useful in the VLSI CAD domain where we have real physical constraints. For example, a particular set of features in a design may warrant the prediction of a certain layout concept with a high certainty factor. However, due to a real physical limitation, this concept is not possible. Therefore, it would be useful to have a "second choice" layout concept. This alternative layout concept would come from another rule based on the same set of features. This layout concept would have a lower certainty factor, but could possibly satisfy both the rules and the physical limitations.

There is another reason closely related to this one. In this research, we found that an incorrect prediction may be "close enough" for estimation purposes. An area or delay estimate based on an incorrect prediction may have only a small percent

difference from the same estimate based on the correct prediction. This means that a "black-and-white" prediction scheme is not necessary. This "gray" area allows more flexibility in the learning system. In addition to satisfying physical limitations, alternative choices for layout concept predictions could yield end results within the bounds of an acceptable error. Only production rule based systems allow alternative predictions. In addition to all these reasons, the production rule based learning system we selected for this research (see Section 2.3.3.1) has already demonstrated usefulness as a data analysis tool [53, 54].

3.2 Overview of a Machine Learning Approach

Our approach to modeling layout tools uses a machine learning system to learn what features of a design relate to certain layout concepts, such as relative placement, for a target layout tool. This mapping of design features to layout concepts is represented by a set of rules. These rules constitute the model, and when applied to a target design, these rules map the design's features into predicted layout information so we can derive estimates of area and delay.

Figure 4 shows a high-level overview of the modeling process. The target layout tool produces layouts for a set of training designs. Topological and graphical features from the training designs and relative placement concepts from the associated layouts are built into training sets. The learning system learns from these training sets a generalized set of rules. These rules constitute the model of the relationships that result from using the layout tool.

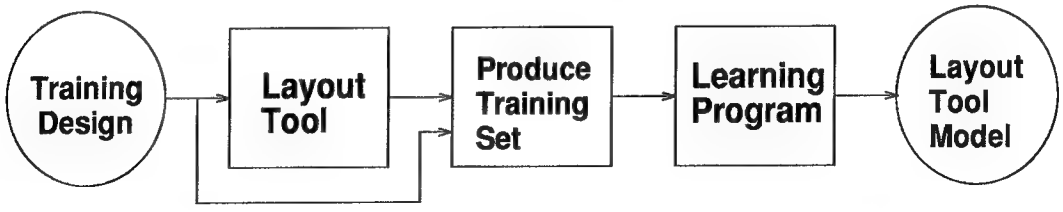


Figure 4 Modeling Process Overview

When applied to a target design, the layout tool model (the set of rules defining the relationships) determines what layout concepts result from the features in the

design. We can use these layout concepts to derive area and delay estimates. A high-level overview of this process is shown in Figure 5. The area and delay estimates are accurate because the rules are based on how the layout tool produces layouts for the training designs. The estimates are obtained more efficiently because it takes much less time to apply the rules than to produce an actual layout, especially for large designs. (The target layout tool is modeled off-line so it does not directly affect the design time.)

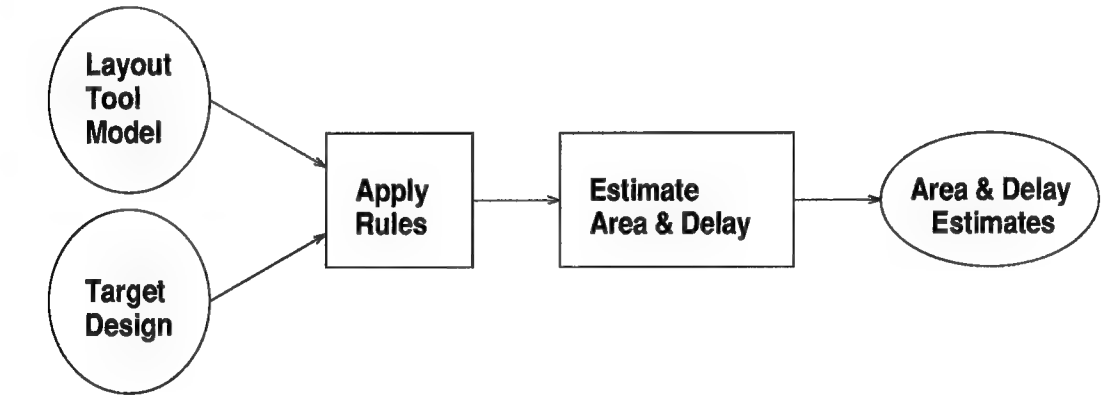


Figure 5 Applying the Model Overview

3.3 Solution Architecture

This section describes the solution architecture for our method of modeling the relationships that result from using a layout tool so we can quickly obtain accurate area and delay estimates. Here we present a high level algorithm that describes how these relationships are modeled and how the model is used to produce area and delay estimates. We describe each step of the algorithm, using implementation details from modeling an actual layout tool to clarify the discussion. The target layout tool for this research is ArtistII from the Keystone Tool Suite [28, 40]. The complete solution architecture as implemented for modeling ArtistII is shown in Figure 6.

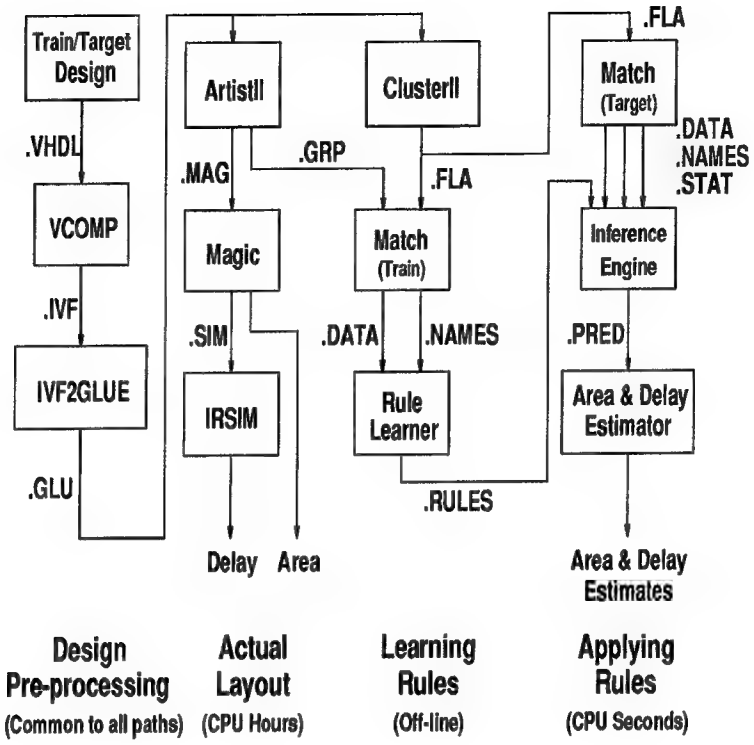


Figure 6 Solution Architecture

3.3.1 Modeling Method Algorithm

There are two major parts to the algorithm: producing the model and using the model. Producing the model (as shown in Figure 4) uses a machine learning system as a data analysis tool to search through a set of training data looking for trends and characteristics that determine the relationships from using the target layout tool. Using the model (as shown in Figure 5) involves applying the model to a target design to get predicted layout information and then using this information to derive area and delay estimates.

3.3.2 Part One: Producing the Model

The first major part of the algorithm consists of two procedures: building a training set and analyzing the training set.

3.3.2.1 Procedure One: Building a Training Set. In this procedure, the user selects a set of designs and processes them through the target layout tool into actual layouts. Information from the training designs and their corresponding layouts is processed into a training set. This training set represents the trends and characteristics that the target layout tool exhibits on the training designs. For this application, a training set must identify design features that affect the layouts. It must also include sufficient information so it can adequately represent the expected target designs.

If the target layout tool is reconfigurable and/or has different execution parameters, the user needs to build different training sets. The same training designs may be used, but different layouts are produced using the target layout tool in the desired configurations with the desired execution parameters. Figure 7 shows the five steps in building a training set.

Step 1. Select set of training designs.

Step 2. Pre-process each training design into a data format that the target layout tool can handle.

Step 3. Use target layout tool to produce a layout for each training design.

Step 4. Transform data from each training design and its layout into training set format and add to training set.

Step 5. (Optional) Repeat Steps 1 through 4 if target layout tool has different configurations and/or execution parameters.

Figure 7 Building a Training Set

Step 1. The first step in building a training set is to select a set of training designs. We use these designs to determine the characteristics of the layout tool being modeled. For this research, the target designs are microprocessor ALU components, so the set of training designs consist of functional units that are commonly found in ALUs (i.e. adders, multipliers, subtracters, absolute value operators, incrementers, decrementers, etc.).

A training set is considered complete when it reaches a point where a substantial increase to its size does not increase the performance of the resultant model (set of rules) [52, 56]. (This performance is measured by applying the model and comparing the results to those from using the real layout tool.) In the context of this research, a substantial increase means adding more training designs to the training set. The appropriate number of training designs is determined through experimentation and analysis of the results. However, the initial training set should include a sufficient representative sample of designs that the model is expected to process. For example, one of the initial training sets for the ArtistII layout tool consists of five adder designs, each using identical architectures, but with different bit widths (4, 8, 12, 16, and 32-bits). This training set is sufficient for adder-based components because it consists of adder designs with the most commonly used bit widths and one with an unusual bit-width.

To make the method as general as possible, the training designs should be generated at a high level so they can be translated into whatever intermediate form of representation is appropriate for the target layout tool. The training designs should also be independent of the layout tool being modeled. In addition, the overall functionality of the training designs should not depend on what node types the target layout tool can handle. (In the context of this research, a node in a design can be either a primitive gate or a standard cell. It is not necessary to make a distinction between these two because this is a general method that deals at a high level where a primitive gate is treated the same as a standard cell.) Also, the training designs themselves should only deal with functionality, degree, and connectivity for each node and interconnections among the nodes. However, only those node types that the target layout tool can process can be used to implement the training designs. This means that each training design must be representable as a netlist of node types

that the target layout tool can process.

Step 2. The second step is to process the training designs into a format that the target layout tool can handle. This can be done by the high level synthesis system that uses the target layout tool. This avoids the necessity to create a new tool.

The target layout tool for this research, ArtistII, requires representation of designs in the GLUE [28, 39] format. This was done by using tools from the Keystone Tool Suite. As shown in Figure 6, the training designs, written in VHDL [61, 62, 63], were translated into the GLUE format by compiling them using the tool called VCOMP [40]. VCOMP compiles the designs into the Intermediate VHDL Format (IVF). Another tool, IVF2GLUE [28, 64], was used to translate this IVF format into the GLUE format.

Step 3. The third step is to process each training design through the target layout tool into an actual layout. If the target layout tool is reconfigurable and/or has user set parameters, these should be set to the desired values that the model is to reflect. A training set and the resultant model can only reflect one configuration of the layout tool using one set of parameter values. (If this method is used to model a target layout tool for a range of configurations and parameter settings, then a super set of training sets should be produced. This will allow the production of different models. Then, we can select the appropriate model to represent the target layout tool based on the desired configuration and/or parameter settings.)

For this research, we used ArtistII to process each training design into an actual layout. As described in Section 2.2.2, an ArtistII layout is represented by a grouping description (.grp) file that indicates the row and column location of each primitive gate in the layout. The gates' relative placement information is obtained from this grouping description file.

Step 4. The fourth step involves processing the training design and the layout into a training set data format. This format is determined by the analysis tool. For this research, the analysis tool is a machine learning system. This learning system determines if there are relationships between the training designs and their corresponding layouts. Another software tool called Match (written exclusively for

this research) processes information from a flattened description file and the layout placement file into the appropriate training set data format.

Each training set includes a set of attribute-value pairs describing various features about the training design and a set of layout concepts that indicate relative placement of the nodes in the layout. The learning system is expected to learn what relationships exist between the training design features and the layout concepts.

As described earlier, the training set must identify design features that affect the layouts so we can learn what relationships exist between them. To do this, we used graph application principles to determine what features to describe. Therefore, the attribute-value pairs describe features dealing with connectivity and degree for each node, and topology for the rest of the design in relation to the node. Attribute-value pairs also describe both individual and relative features of the nodes. Individual node features include the minimum area, height, and width of each node. They also include the associated delays for rising and falling output transitions, the number of inputs, and the node's function. Relative node features are explained below. (Note: all of this information is obtained from a node library as the training set is built.) While some of the features seem redundant, such as area when height and width are available, it is necessary to explicitly describe all features. This is because the learning system is incapable of learning relationships among the features. It is an inductive learning system and can only learn by observation.

Because it is necessary to pair up each node with every other node in order to obtain relative placement information from the layout, node pairs are used to form the individual training instances in the training set. This requires pairing up each node with every other node in the training design. This node pairing produces a training set of $[n^2-n]/2$ pairs where n is the number of nodes. While this is $O(n^2)$, it can be reduced somewhat because it is meaningless to pair up a node with itself (the $-n$ factor), and mirrored pairings provide the same data (division by 2). The features for each node pair include each node's individual set of attribute-value pairs as described previously, plus a set of relational attribute-value pairs. These relational features describe how the two nodes relate to each other. This includes information about which node has more area, more height, more width, more delay, more total inputs, more external inputs, and more internal inputs.

Other relational features include whether or not one node has an input to, output from, or no connection to the other node, and if the two nodes share a common destination or a common source. Once again, it is necessary to explicitly describe these relationships between the two nodes because the learning system is incapable of learning them.

We can determine if the list of training design features is complete by using the same test as the training set. If the list is complete, the performance of the system is not increased by adding more features. This is also determined through experimentation. For this research, the initial list of features came from the CAD domain. The connectivity and topology features are based on relevant features found in graph applications. The physical features deal with the main goal of the research: estimating area and delay. (Our goal for the initial list is to include as much descriptive information as possible. Then we can prune the information that does not show up in the relationships.)

The layout information (relative placement) for each node pair is obtained by examining the layout to determine where the nodes are located relative to each other. For example, they may be located next to each other, at opposite ends of the layout, or somewhere in between. Each node pair has a relative placement as its concept, and the learning system tries to learn what features about the two nodes determine where they are placed relative to each other. We can use the relative placement to derive the relative distance. The relative distance determines wirelength between the logically connected nodes.

The relative placement concepts we use are classifications based on the Manhattan distance between the two nodes in the node pair. These classifications could include only one distance or more than one, depending on how well the resultant model performs. However, the same distance may not be used by more than one concept. The learning system requires discrete categorization and overlapping distances perturbes the learning process. (This is explained more in Section 4.1.)

For this research, the Match program parses through each training design's flattened description (.fla) file and the corresponding layout placement (.grp) file to get the raw training data. As shown in Figure 6, the flattened description file is produced

by a tool called ClusterII [40, 41]. The primary purpose of this tool is producing an initial gate clustering that helps speed up and improve the layout processing done by ArtistII. However, for this research, ClusterII is used only to produce the flattened file description, and not for initial gate clustering. (If we were to use ClusterII to provide an initial clustering, the resultant layouts would be different and the learning system would learn relationships due to a combination of ArtistII and ClusterII. We only wished to model ArtistII, so we did not use ClusterII in the clustering mode.) Once all training designs and layouts have been processed into the proper data format, all of the data is accumulated into one training set.

However, before the learning system can learn rules from the training set, the Match program must build one more file. This file is called the Partial Domain Model (PDM) [53, 54]. (This domain information is contained in the .names file shown in Figure 6.) The purpose of the PDM file is to describe the target domain so the learning system has a context in which to operate. The PDM file does this by listing the concepts we want the learning system to learn and by listing the relevant features that describe our domain. (These are the same concepts and features as described above.)

Because this file can not have an infinite size, it can only describe part of the domain, hence the name "Partial" Domain Model. In addition, the PDM should not be confused with the target layout tool model, which is a set of rules. The PDM file can be as simple or as complex as necessary. The complexity depends on the relevant features in the domain and the desired concepts to be learned. The PDM file provides a powerful way to generalize the learning system so it can learn in any domain. Figure 8 shows one of the PDM files used in this research. This file lists the layout concepts that we expect the learning system to learn on the first line. The design features are listed on the remaining lines. This file also describes the type (symbolic, numeric) and admissible values for each feature in the training set.

For example, this particular PDM file shows seven layout concepts: *near_s, near_d, close_1, close_2, far_1, far_2,* and *way_out.* (The reasons for these particular concepts are explained in Section 4.1.) These concepts represent relative placements of the two nodes in each training instance. The relative placement of two nodes indicates

how far apart the nodes are. Table 3 shows the seven relative placement concepts and the corresponding number of positions apart. The concept of *near_s* indicates that the two nodes are placed next to each on the same row. The concept of *near_d* indicates that the two nodes are next to each other, but on different rows. In other words, one node is above and one node is below. The concept of *close_1* indicates the two nodes are two or three positions apart from each other, on the same row or on different rows. The concept of *close_2* indicates the nodes are four or five positions apart. The concept of *far_1* indicates the two nodes are six or seven positions apart. The concept of *far_2* indicates the two nodes are eight or nine positions apart. The concept of *way_out* indicates the two nodes are 10 or more positions apart.

Table 3 Relationship of Node Position to Relative Placement

Relative Placement Concept	Number of Positions Apart
near_s	1 (same row)
near_d	1 (different row)
close_1	2-3
close_2	4-5
far_1	6-7
far_2	8-9
way_out	10+

The remainder of the file shows the design features and their types and admissible values. For example, the design feature *log_conn* indicates if the two nodes in the training instance are logically connected. The admissible values are either *yes* or *no*. (The admissible values also imply the type; in this case *yes* and *no* indicate a symbolic type.) Another feature such as *area_1*, which indicates the area required by the first node in the node pair, has a continuous admissible value. This means it is a numeric type and can take on any numerical value. Once the training set and PDM files are complete, the learning system can begin analyzing the training set in order to produce rules.

Step 5. The user may repeat steps 1 through 4 if the target layout tool has different configurations and/or different execution parameters and these need to be modeled. The user repeats steps 1 through 4 for each desired configuration of the

```
near_s, near_d, close_1, close_2, far_1, far_2, way_out. | concepts
log_conn:       yes, no.          com_dest:       yes, no.
com_src:        yes, no.          a1_a2:          >, =, <.
h1_h2:          >, =, <.          w1_w2:          >, =, <.
ni1_ni2:        >, =, <.          nx1_nx2:        >, =, <.
nt1_nt2:        >, =, <.          fun1_fun2:      ==, !=.
area_1:         continuous.       height_1:       continuous.
width_1:        continuous.       delay_1:        continuous.
funct_1:        ai2,ai3,ai4,aoi21,aoi211,aoi22,i1,
oai21,oai211,oai22,oi2,oi3,oi4. inputs_1:       continuous.
ext_1:          continuous.       int_1:          continuous.
1lev1_1:        continuous.       1lev2_1:        continuous.
1lev3_1:        continuous.       1lev4_1:        continuous.
2lev1_1:        continuous.       2lev2_1:        continuous.
2lev3_1:        continuous.       2lev4_1:        continuous.
area_2:         continuous.       height_2:       continuous.
width_2:        continuous.       delay_2:        continuous.
funct_2:        ai2,ai3,ai4,aoi21,aoi211,aoi22,i1,
oai21,oai211,oai22,oi2,oi3,oi4. inputs_2:       continuous.
ext_2:          continuous.       int_2:          continuous.
1lev1_2:        continuous.       1lev2_2:        continuous.
1lev3_2:        continuous.       1lev4_2:        continuous.
2lev1_2:        continuous.       2lev2_2:        continuous.
2lev3_2:        continuous.       2lev4_2:        continuous.
```

Figure 8 Example Partial Domain Model

target layout tool using a particular set of parameter values. Only one configuration using one set of parameters may be represented by a single training set. Therefore, different training sets are required to produce different models. Each model represents the same target layout tool, but for different configurations using different parameter values.

For example, one of the validation designs for this research is a 16-bit multiplier (see Section 5.2). This design consists of about 3000 primitive gates. We discovered that ArtistII required more than a couple of weeks to produce a layout for this design using 10,000 iterations. (The number of iterations is one of the execution parameters for ArtistII. Larger numbers of iterations produces smaller layouts.) We also found that running ArtistII continuously for two or more weeks was difficult to achieve

on time-shared computing systems that reboot weekly. Therefore, it was necessary to reduce the number of iterations to 100 in order to get a layout for comparison purposes. However, if we used a training set based on 10,000 iterations to produce a model, and we used this model to process the 16-bit multiplier, the resultant layout should not be compared to an actual layout generated by ArtistII using only 100 iterations. The comparison would be invalid. Therefore, a new training set consisting of the same training designs but with actual layouts generated by ArtistII using only 100 iterations was necessary to train the learning system to produce a new model. Using this new model on the 16-bit multiplier provided a valid test.

3.3.2.2 Procedure Two: Analyzing the Training Set. This procedure has three steps. The training set is analyzed for relationships between design features and layout concepts, any relationships found are put into a list, and if no relationships are found, the process is repeated using different analysis tool parameters. Figure 9 shows these steps.

Step 1. Analyze training set data to determine if there are any relationships between training design features and corresponding layout concepts.

Step 2. If relationships are found, accumulate them into a list. This list constitutes the model of the target layout tool.

Step 3. If no relationships are found, use different analysis parameters and repeat Step 1. If no relationships are found after several tries, use a different training set.

Figure 9 Analyzing the Training Set

Step 1. The first step in analyzing the training set data is to determine if there are any relationships between features in the training design and the corresponding layout concepts. These relationships indicate trends and characteristics of the target layout tool. For example, if the target layout tool tends to place logically connected nodes

close together, this will come out in the analysis. It will show up as a relationship between logically connected nodes and close placement.

This analysis is performed by the machine learning system. The learning system searches through the training set data looking for evidence that supports relationships between the design features and the relative placement concepts. If the evidence for a particular relationship is strong enough, a rule describing that relationship is produced. The necessary strength is determined by a parameter set by the user before executing the learning system. (This and other aspects of the learning system in analyzing the training set are described more in Section 4.2.)

Step 2. The second step is to produce a list of relationships that exist. The learning system does this by producing a set of rules that formally and logically describe the relationships. As described in Section 2.3.3, these rules are of the form $< lhs > \rightarrow < rhs >$, where $< lhs >$ is a conjunction of features (attribute-value pairs) and $< rhs >$ is the layout placement concept that the learning system determines is related to the features in $< lhs >$. An example set of rules is shown in Figure 10. Because these rules are based on relationships between features in the training designs and corresponding layout concepts, they reflect the characteristics of the target layout tool. When applied to a target design, these rules map features in the design to layout concepts and provide layout information that can be used to estimate area and delay.

For example, the first rule maps a relative placement concept of *near_d* to node pairs that have a different number of external inputs. The statistics included in this rule show that 64 training instances indicate the concept of *near_d*. Out of these training instances, 31 instances indicate the node pairs have a different number of external inputs. This works out to a 48% belief in the rule. On the other hand, the statistics also show that 5872 training instances have a relative placement concept other than *near_d*. Out of these instances, 1099 instances indicate the node pairs have a different number of external inputs. This works out to a 19% disbelief in the rule. All of this generates a certainty factor of 0.710, which is explained next.

As shown in Figure 10, each rule has a certainty factor associated with it. (Certainty factors were introduced in Section 2.3.) The original certainty factors (cf) used by the Rule Learner system were generated by Equation 3-1, where p is the

```
(nx1_nx2 !=)  ==> near_d
 { pos=  0.48,  neg=  0.19,  cf=  0.710
  p=31, n=1099, tp=64, tn=5872 }

(nx1_nx2 !=)  ==> near_s
 { pos=  0.41,  neg=  0.18,  cf=  0.692
  p=109, n=1021, tp=266, tn=5670 }

(ni1_ni2 ==)  ==> way_out
 { pos=  0.38,  neg=  0.17,  cf=  0.692
  p=1450, n=356, tp=3826, tn=2110 }

(fun1_fun2 ==)  ==> way_out
 { pos=  0.32,  neg=  0.16,  cf=  0.669
  p=1220, n=332, tp=3826, tn=2110 }

(a1_a2 ==) (h1_h2 !=)  ==> near_s
 { pos=  0.70,  neg=  0.39,  cf=  0.640
  p=186, n=2210, tp=266, tn=5670 }

(a1_a2 ==) (h1_h2 !=)  ==> near_d
 { pos=  0.64,  neg=  0.40,  cf=  0.607
  p=41, n=2355, tp=64, tn=5872 }

(com_src yes)  ==> close_1
 { pos=  0.14,  neg=  0.09,  cf=  0.605
  p=75, n=475, tp=546, tn=5390 }
```

Figure 10 Example Set of Rules No. 1

number of examples in the training set supporting belief in the rule (called *positive examples*), and n is the number of examples supporting disbelief in the rule (called *negative examples*).

$$cf = \frac{p}{p+n} \tag{3-1}$$

However, due to the large inequality of the number of training instances for each class, Equation 3-1 did not produce a good measure of a rule's certainty. Through experimentation with many training sets from this domain, we found that the total

number of negative examples was usually much larger than the total number of positive examples. This tended to force the certainty factors downward by causing n to be much larger than p. But, if we look at the ratio of the number of positive examples of the rule to the total number of positive examples and the ratio of the number of negative examples of the rule to the total number of negative examples, we can obtain a better perspective of the rule's certainty.

We obtained a better perspective of a rules's certainty by using Equation 3-2, where p and n are the same as before, tp is the total number of examples having the same concept, and tn is the total number of examples not having the same concept.

$$cf = \frac{\frac{p}{tp}}{\frac{p}{tp} + \frac{n}{tn}} \tag{3-2}$$

This formula normalizes the certainty factors by dividing the positive and negative examples of the rules by their respective total number of examples. This reduces the effects that a very large total number of positive or negative examples have on the certainty factors. Further examination of this formula shows that when the ratios of p/tp and n/tn are equal in value, a certainty factor of 0.5 is generated. This means that the rule has a 50% chance of being true or false. This makes sense when we consider a certainty factor above 0.5 as showing support for belief in the rule, and a certainty factor below 0.5 as showing support for disbelief in the rule.

We developed this method for calculating the certainty factor to accommodate limitations in this domain, namely the highly unequal distribution of positive and negative examples. Therefore, as in other applications [49, 53], these certainty factors should not be viewed as probabilities of a rule's truth. Rather, these certainty factors should be used to rank order the rules and to weight the rules as they are used to classify the test cases.

When more than one rule predicts the same concept for a node pair, the certainty factors are combined into one factor [49]. If other rules predict another concept for the same node pair, those rules' certainty factors are combined. This is repeated until each concept that results from this same node pair has only one certainty factor associated with it. These combined certainty factors are used to rank order the predicted concepts for that node pair. The concept with the highest combined certainty factor

becomes the "first choice" for that node pair. The concept with the next highest combined certainty factor becomes the "second choice", and so on.

The certainty factors are combined according to Equation 3-3, where X and Y are the two certainty factors we wish to combine (order does not matter). If we wish to combine more than two certainty factors, we start out by combining the first two. This combined certainty factor is then combined with the next certainty factor, and so on. This equation prevents certainty factors from combining to a value larger than one. However, the combined certainty factor will approach one eventually, given enough certainty factors.

$$cf_{comb}(X,Y) = X + Y(1 - X) \tag{3-3}$$

Also, the learning system was originally set up to use positive and negative thresholds as search parameters. Only the rules that had p/tp above a certain threshold (user-defined at runtime) and n/tn below a certain threshold (also user-defined) were produced. Using this method in the VLSI CAD domain produced very few good rules. However, by using the certainty factors as search parameters (produce only those rules with a certainty factor above a certain user-defined threshold), more good rules were produced. In addition, by setting the certainty factor threshold high, more conjuncts were used in the left-hand side of the rules and provided more information on how the design features map to the layout concepts.

Step 3. If no relationships are found, the third step is to repeat Step 1 using different analysis parameters. For the machine learning system, this means using different biases [53] and learning again from the same training set. These user-specified biases include the desired range of certainty factors for acceptable rules, the maximum number of features that may appear in the left-hand side of a rule, and the beam width for the beam search (see Section 2.3). The certainty factor range determines the upper and lower limits on certainty factor values. Only those rules whose certainty factors fall within this range are accepted.

The biases should be thought of as filters. They may be adjusted to filter out relationships based on certainty factors or the number of features required in the left-hand side or adjusted to allow more relationships through. For example, increasing

the beam width allows the search routine to open up more nodes and find more relationships. If rules are not found using a certainty factor range of 0.8 to 0.9, then maybe using a range of 0.7 to 0.9 would produce better results. In addition, increasing the maximum number of features that may appear in the left-hand side will generate more rules. Of course, all of this depends on whether or not the relationships exist.

3.3.3 Part Two: Using the Model

The set of rules produced by the learning system makes up the model of the relationships from using the target layout tool. To apply the model, the target design must be pre-processed in the same fashion as a training design. This is shown in Figure 6. However, a layout is not produced. Instead, the model determines what layout concepts result from certain features in the design. These concepts indicate the relative placement of each node pair in the target design. This relative placement information is then provided to our estimator so it can derive area and delay estimates.

3.3.3.1 Procedure Three: Applying the Model. As shown in Figure 11, there are three steps in this procedure. The target design is pre-processed, the model is applied, and if the model produces no predicted layout results, a different model is selected and the process is repeated.

Step 1. The first step is to pre-process the target design into the proper format. This format is identical to that for a training design, but because no layout is produced, no relative placement information is included. Pre-processing the target design into this format makes it straightforward to match features from the target design to those in each of the rules.

Step 2. The second step searches through the features of each node pair in the target design data to determine if any features match those in the set of rules. The set of rules are listed in descending order according to their certainty factors. The target design data is searched first using the rule with the highest certainty factor. The data is searched again using the rule with the next highest certainty factor, and so on. Certainty factors are combined as necessary (as described in Section 3.3.2.2) until

Step 1. Pre-process target design into proper format.

Step 2. Apply the model by searching the target design data to determine if any target design features match features in the list of relationships. If matches are found, add the corresponding layout concepts to the predicted layout information.

Step 3. If no matching design features are found, select a different model and repeat Step 2.

Figure 11 Applying the Model

each node pair has a list of predicted concepts rank-ordered according to their final combined certainty factor. One of the main procedures in the Rule Learner system is called the *Inference Engine* [54]. The Inference Engine applies the rules to the target design data and produces a file containing all node pairs and their predicted relative placement concepts.

Step 3. This step is executed if no matches are found. A different model of the same target layout tool is selected and Step 2 is repeated. Of course, this requires different models to have been produced already. Because different models represent different configurations of the target layout tool, using a different model is the same as using the layout tool in a different configuration. Therefore, the selection of a different model is based on what target layout tool configuration would produce better results. As it is with actual layout tools, some models may not produce satisfactory layouts for certain designs.

3.3.3.2 Procedure Four: Estimating Delay. Using the layout information (relative placement) obtained by applying the rules to a target design, the delay estimator finds the design's critical path and calculates the worst case delay. The steps are shown in Figure 12.

Step 1. Use predicted layout information to produce a set of estimated wirelengths between all logically connected nodes.

Step 2. Perform critical path analysis based on critical path's output node having a low-high transition.

Step 3. Repeat Step 2 based on the critical path's output node having a high-low transition.

Figure 12 Estimating Delay

Step 1. The first step in estimating delay is to derive the necessary wirelengths from the layout information. We can derive the relative distance between all logically connected node pairs from their predicted relative placement. This relative distance is translated into an estimated wirelength. For example, a predicted relative placement concept of *near_s* means the two nodes are next to each other on the same row. This translates into a relative distance of one unit. For Magic files produced by the ArtistII layout tool using a two micron process, one unit of distance translates into about 70 microns of wirelength. We found this average unit distance by performing a statistical analysis of the distance between node pairs that were next to each other for many test and training design layouts.

Steps 2 and 3. The second and third steps determine the critical path delay by performing an exhaustive depth-first search of all possible input-to-output paths in the target design. Step 2 does this for a high-low transition on the path's output node while Step 3 does this for a low-high transition on the path's output node. The internal delay for each node along a path is obtained from the library file and the wirelength between nodes is known from Step 1. The wirelength determines the wiring delay. We used delay equations from research on the TinkerTool [65] system to calculate the wiring delay. The internal node delays and the wiring delays are summed along each path to produce that path's overall delay. The path with the worst delay is considered the critical path. The delay of this path is considered the worst case delay for the target design, based on either a high-low or low-high

transition on the critical path's output.

Of course, performing an exhaustive search of all possible input-to-output paths is time-consuming for large designs. However, we have shown that we can pre-process the target design, apply the model, and estimate both area and delay, all in a time much faster than just producing an actual layout.

3.3.3.3 Procedure Five: Estimating Area. Using the layout information (relative placement) obtained by applying the rules to a target design, the area estimator calculates the overall area requirement for the design. The steps are shown in Figure 13.

Step 1. Use predicted layout information to produce an estimated layout.

Step 2. Estimate height and width of estimated layout due to individual nodes only.

Step 3. Add in estimated height and width of wiring area to produce total area estimate.

Figure 13 Estimating Area

Step 1. The first step in determining the overall area is "growing" an estimated layout. One of the nodes is selected as the "seed" node. This node is placed in the middle of the estimated layout. Other nodes are placed in this layout according to their predicted relative placement. Once all nodes have been placed, we continue with Step 2. This is very different from classical placement routines because there is no analysis of the target design. The only information available is the relative placement of the node pairs.

Step 2. The second step is determining the contribution of only the nodes (no interconnects) to the overall height and width. We obtained the height and width of each node from the node library file. The height for each row in the estimated

layout is found by using the value of the tallest node in the row as the height of that row. We found the overall width by adding up the node widths in each row in the estimated layout and using the width of the widest row.

Step 3. The third step is adding in the area due to the wires. A very simple formula based on the number of rows and columns in the design was used to calculate an estimate of the required wire area. More rows required more horizontal wires and more columns required more vertical wires. This formula produced estimates within 20%. (The main intent here is to show that it is possible to derive good area estimates using machine learning techniques. We expect that better models of wire area will produce better results, but the learning results will be the same.)

3.4 Summary

This chapter first presented some alternative methods to modeling layout tools. It then showed how modeling the input-to-output relationships that result from using layout tools was the best choice. This chapter went on to describe a high-level overview and solution architecture for this method. This chapter described the solution architecture as a generic algorithm. This algorithm uses a machine learning system to model the relationships and to apply the model to target designs. To clarify the discussion, this chapter also described some examples from using this method to model an actual layout tool. However, it should be noted here that the intent of this chapter was to introduce and describe our modeling method at a high level without going into detail. The next chapter expands this discussion of our method by describing an actual implementation in detail.

4.0 AN IMPLEMENTATION USING MACHINE LEARNING

This chapter describes in detail an implementation of the area and delay estimation method. It does this by following the solution architecture and algorithm described in the previous chapter. This chapter shows how this method is used to model an actual layout tool using real training designs and real test designs. The entire method is described from beginning to end, starting with building a training set and finishing with estimating area and delay. This chapter also discusses the reasons behind the implementations chosen for each of the procedures in the method.

Section 4.1 discusses the first procedure in using machine learning techniques to model the input-to-output relationships from using layout tools: building a training set. This section discusses the philosophy behind training sets by describing their function and basic requirements. This section also gives an example of how an actual training set is built using real training designs. Section 4.2 describes how the learning system analyzes a training set and produces a set of rules that models the target layout tool's relationships. This section also describes the user-specified biases for the learning system. An example of using the learning system to analyze an actual training set accompanies and illustrates these descriptions. Section 4.3 describes how the model (set of rules) is applied to real designs. This section shows how and why the set of rules maps the relevant design features into layout concepts. To clarify the discussion, this section shows how the set of rules are applied to an actual test design to produce a set of layout concepts. Section 4.4 describes how the estimator uses these layout concepts to provide delay estimates. This section shows how the layout concepts provide wirelength information so the estimator can calculate the delay between gates in the design. This section also describes how the estimator performs a search of all of the input-to-output paths in a real test design looking for the critical path. Delay estimates for the five test designs are presented along with their percent differences from the actual delay values. Section 4.5 describes how area estimates are derived from the layout concepts. This section first shows how the estimator uses layout concepts to generate an estimated layout. Then it shows

how the estimator uses this estimated layout to produce height and width estimates. This is illustrated by using the estimator on a set of layout concepts from an actual test design. The height and width estimates for each of the test designs are presented along with their percent differences from the actual height and width values.

The target layout tool in all of these examples is the ArtistII layout tool. This tool is described in Section 2.2.2. The example training set is built using five training designs: a 4-bit adder, 8-bit adder, 12-bit adder, 16-bit adder, and 32-bit adder. The test designs we describe in detail to illustrate applying the model include a 4-bit adder, 12-bit adder, 24-bit adder, 16-bit absolute value operator, and 4-bit multiplier. The test designs include adders because the training set is built from adder designs. The other test designs, the absolute value operator and multiplier, are used to show how an adder-based training set can work for other types of designs.

4.1 Building a Training Set

A training set forms the foundation for the set of rules that models the target layout tool. The learning system used in this research, due to its general nature, also requires a description of the domain to provide a context to the information in the training set, but only the information in this training set is used to produce a set of rules. Therefore, the training set must be complete in that it provides enough information so a representative model is generated. If it were possible to have an unlimited training set where all possible training instances were included, and all possible information about the designs were included in the training instances, then the set of rules would provide a perfect model. However, in the real world, perfection is not possible because of natural limitations. It is not possible to have an unlimited training set that includes all information on the designs. It is possible, on the other hand, to have a training set that adequately describes a sufficient sample set of designs. The training instances include enough information about the features to yield accurate rules and the training set has enough training instances to yield a sufficient set of rules. At this point the training set is said to be complete.

This research uses the performance of the resultant model as a measure of training set completeness. (This is briefly described in Section 3.3.2.1.) A training set is

considered complete when a substantial increase to its size does not increase the performance of the model. (Here we define size as a function of the amount of information in each training instance and the total number of training instances.) An initial training set is built and the performance of the model on test designs is measured. Then more information is added to the training set and another model is built. The performance of this model is measured and compared. This process is repeated until the performance of the model starts to decrease or stops increasing.

For this research, we decided to start out using as much relevant information as possible in each training instance. The purpose of a training instance is to describe various features about an entity and to show the correct classification of that entity. In this research, the entity is a pair of gates and the features describe the individual gates in the pair and their relationships to each other. The features themselves are based on graph features such as connectivity, topology, and individual node attributes. The Partial Domain Model (PDM) shown in Figure 14 shows all of the relevant features. (Figure 14 is the same as Figure 8, but is shown here again for easy reference.) Because we are interested in learning layout concepts for the relative placement of two gates, each training instance describes features about the individual gates and how they relate to each other. Therefore, this initial PDM lists features that describe individual node attributes and relational attributes between the two gates.

The first line shows the seven relative placement concepts we want the learning system to learn about. (These relative placement concepts are based on the Manhattan distance between gates.) These are the same concepts described in Section 3.3.2.1. More concepts could be used to handle up to several hundred positions if necessary. However, when the learning system has to learn a large number of concepts, the learning process degrades considerably.

To improve the learning process, it is better to learn fewer concepts. Fewer concepts means there are more training instances for each concept. More training instances provide more evidence for either supporting belief or disbelief in the rule. More evidence improves the learning results. Having fewer concepts also means that each concept must have a wider scope in order to cover the same number of cases. For example, we could use 20 concepts to cover 20 relative placements with each concept covering only one relative placement (i.e., a concept of *apart_5* could indicate the two

```
near_s, near_d, close_1, close_2, far_1, far_2, way_out. | concepts
log_conn:      yes, no.          com_dest:      yes, no.
com_src:       yes, no.          a1_a2:         >, =, <.
h1_h2:         >, =, <.          w1_w2:         >, =, <.
ni1_ni2:       >, =, <.          nx1_nx2:       >, =, <.
nt1_nt2:       >, =, <.          fun1_fun2:     ==, !=.
area_1:        continuous.       height_1:      continuous.
width_1:       continuous.       delay_1:       continuous.
funct_1:       ai2,ai3,ai4,aoi21,aoi211,aoi22,i1,
oai21,oai211,oai22,oi2,oi3,oi4. inputs_1:      continuous.
ext_1:         continuous.       int_1:         continuous.
1lev1_1:       continuous.       1lev2_1:       continuous.
1lev3_1:       continuous.       1lev4_1:       continuous.
2lev1_1:       continuous.       2lev2_1:       continuous.
2lev3_1:       continuous.       2lev4_1:       continuous.
area_2:        continuous.       height_2:      continuous.
width_2:       continuous.       delay_2:       continuous.
funct_2:       ai2,ai3,ai4,aoi21,aoi211,aoi22,i1,
oai21,oai211,oai22,oi2,oi3,oi4. inputs_2:      continuous.
ext_2:         continuous.       int_2:         continuous.
1lev1_2:       continuous.       1lev2_2:       continuous.
1lev3_2:       continuous.       1lev4_2:       continuous.
2lev1_2:       continuous.       2lev2_2:       continuous.
2lev3_2:       continuous.       2lev4_2:       continuous.
```

Figure 14 Initial Partial Domain Model

gates are placed five positions away from each other, where a position is a possible gate location). An alternative is to use fewer concepts, but cover more relative placements with each one. For example, we could cover up to 20 relative placements with only five concepts if each concept covered four relative placements (i.e., a concept of *apart_5.8* could indicate the two gates are placed either five, six, seven, or eight positions apart). We have covered the same number of relative placements, but used only five concepts.

In this implementation, we used seven concepts to cover all of the relative placements. The concept of *near_s* indicates the two gates are located next to each other on the same row. The concept of *near_d* indicates the two gates are next to each other, but on different rows. In other words, one gate is right above or below the

other. The concept of *close_1* indicates the two gates are located two or three positions apart from each other, on the same row or on different rows. The concept of *close_2* indicates the gates are located four or five positions apart. The concept of *far_1* indicates the two gates are six or seven positions apart. The concept of *far_2* indicates the two gates are eight or nine positions apart. The concept of *way_out* indicates the two gates are 10 or more positions apart.

It turns out that by using design features based on graph applications, the gate pairs within 10 positions of each other produce more rules than gates further apart. For example, as described later in this section, the learning system produced 35 rules based on the training set in this implementation. There were 1144 training instances in this training set, with 585 of them having the concept of *way_out*. This means more than half of the training set had relative placements of 10 or more positions apart. However, only eight of the 35 rules predicted a concept of *way_out*. As a comparison, seven rules were produced for the concept of *near_s* based on only 82 training instances. Nine rules were produced for the concept of *close_1* based on 123 training instances. Five rules were produced for the concept of *close_2* based on 130 training instances, and six rules were produced for the concept of *far_1* based on 119 training instances. No rules were produced for the remaining concepts of *near_d* and *far_2* based on a total of 105 training instances. This shows that a large number of training instances supporting a particular concept does not necessarily produce more rules. In addition, because rules are only produced when there is substantial evidence in the data to support the rules, this shows there is more evidence linking the design features to concepts for relative placements less than 10 positions apart. Therefore, we decided to concentrate on learning about the gate pairs that are within 10 positions of each other. The seven concepts handle this sufficiently.

Another reason we can concentrate on gate pairs within 10 positions of each other is that gates further away have very little effect on each other as far as relative placement. Of course, if they are connected, the wirelength would start to get long and the delay of the driving gate would increase. However, using rules to predict the relative placement of gates based on gate pairs with large distances between them is not necessary. In practice, each gate in the pair is usually connected to another gate which is closer. A relative placement based on a pairing with this other gate would

provide better results because the gates are closer and have more effect on each other.

The results in Chapter 5 shows that this approach holds, even for large designs. When gate pairs that exhibit the proper features are placed according to one of these seven concepts, gates that should be further apart end up as such. This approach also allows us to handle large designs as well as small ones.

The ability to represent the necessary layout information with only seven concepts is also an example of how "black and white" classifications are not necessary in this method. Classifying a gate pair as being either two or three positions apart introduces a gray area into the modeling process. However, it turns out the difference between these positions is small enough that it has little effect on the area and delay estimates.

It should be noted here that overlapping the concepts (having more than one concept include the same number of positions apart) will introduce confusion into the learning process. When the learning system analyzes the training set, it needs to have one and only one correct answer for each of the training instances. Having more than one correct answer would produce conflicting rules. In turn, the accuracy of the area and delay estimates would decrease.

The other information in the PDM describes attributes about the two gates in the gate pair. The first 10 features describe how the two gates relate to each other. The feature *log_conn* indicates if the two gates are logically connected. A gate pair is considered to be logically connected if one gate in the pair either provides an input to the other gate or receives an output from the other gate. The feature *com_dest* indicates if the two gates have a common destination (i.e., both gates provide inputs to the same gate). The feature *com_src* indicates if both gates have inputs from a common source (i.e., both gates share the output of the same gate). The features *a1_a2, h1_h2,* and *w1_w2* indicate if the first gate has more, equal, or less area, height, and width than the second gate in the pair. The features *nt1_nt2, nx1_nx2,* and *ni1_ni2* indicate if the first gate has more, equal, or less internal, external, and total inputs than the second gate. The feature *fun1_fun2* indicates if the functions of the two gates are the same or different. Each of these 10 features is a symbolic type. The first three features have the symbols *yes* and *no* as admissible values. The

next six features have the symbols >, =, and < as admissible values. The feature *fun1_fun2* has the symbols == and != as admissible values.

The remaining features describe attributes about the individual gates. Any feature with a *"_1"* extension describes an attribute about the first gate listed in the training instance. Any feature with a *"_2"* extension describes an attribute about the first gate listed in the training instance. The features *area, height, width*, and *delay* indicate the area, height, width, and intrinsic delay of the individual gate. These features are a continuous type. Any real number is admissible. The feature *funct* indicates the function of the individual gate. This feature is a symbolic type with all of the possible primitive gates as admissible values. The features *inputs, ext*, and *int* indicate how many total inputs, external inputs, and internal inputs the individual gate has. These three features are a continuous type.

The next eight features describe the topology of the design in relation to each of the individual gates in the training instance. We wanted to know if gates that are one or two levels away affected the relative placement of the two gates in the gate pair. These topology features are of the form *xlevy* where x is the number of levels away and y is the number of inputs. The continuous values for these features indicate how many gates that are x levels away have y inputs. For example, if the feature *2lev3_1* has a value of five, then two levels away from the first gate in the gate pair there are a total of five gates with three inputs each. The topology description was set up this way so the rules could indicate if there were first or second order effects from other gates in the layout on the two gates in the training instance.

Now that we have described all of the relevant information in a individual training instance, the next consideration in a training set is the number of training instances. The goal here as well is to have a large enough training set so the performance of the model is not increased by a substantial increase to the training set. The initial training set was made up of all possible gate pairs from the five adders used as training designs. Four of these adders have the most common bit widths: 4, 8, 16, and 32. One of these adders has an uncommon bit-width of 12 just to include some evidence for odd designs. Each design requires a different size layout with different physical dimensions. These designs also exhibit a wide variety of the features listed in the PDM. These five designs provided 107,690 training instances. However, the version of

the learning system we were using was unable to handle this many training instances, so we had to remove the largest training design, the 32-bit adder. This reduced the number of training instances down to 25,069. It is better to remove an entire set of training instances based on one training design than to remove only some of them. This is because each training design adds its own unique set of characteristics and this set would be incomplete if all training instances are not included. As it turns out, this reduced number of training instances was more than enough to produce a good model. Later in this section, we describe how this training set was further reduced to an even more manageable size.

The remainder of this section describes how to build a training set by following the procedure and actually building one. The first step in building a training set is selecting a set of training designs. Our example training set now consists of four training designs: a 4-bit adder, 8-bit adder, 12-bit adder, and 16-bit adder. The reasons for these designs is explained in the previous paragraph.

The second step is pre-processing each training design into a format the target layout tool can handle. For the ArtistII layout tool, a design must be in the GLUE format. As shown in the solution architecture in Figure 15, we start out with a VHDL description (.VHDL) of each training design. (Figure 15 is the same as Figure 6. It is copied here for easier reference.) We started with VHDL descriptions because we want the designs to be in a general, technology independent, and layout tool independent format. In fact, we were able to use these same designs as training designs to model the other layout tool, TimberWolf. Each training design is compiled into an Intermediate VHDL Format (.IVF) using VCOMP. A tool called IVF2GLUE translates each of the .IVF files into a GLUE format (.GLU) file. Once all training designs are in the GLUE format, we can proceed to the next step.

The third step is producing actual layouts for each of the training designs. For the training designs in this example, we ran ArtistII using 10,000 iterations. This number of iterations is based on information from the author of ArtistII, R. Owens, for the number of gates in these designs. These are small designs so it did not take too long to get the layouts. The execution times are shown in Table 4. For each training design, ArtistII produced a Magic layout file (.MAG) and a grouping file (.GRP). (The MAGIC layout file is only used for test designs to get the actual area

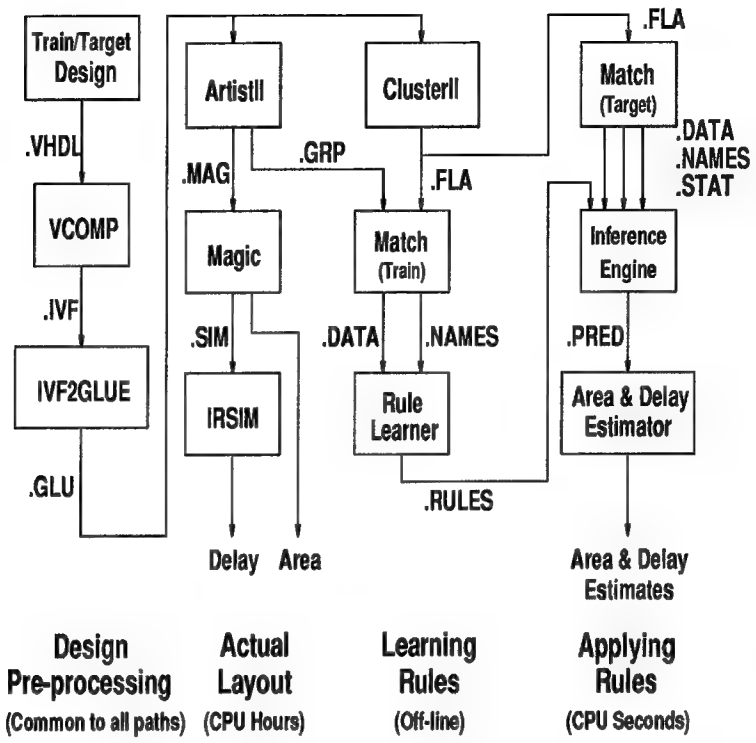


Figure 15 Solution Architecture

and provide electrical information to IRSIM to get the actual critical path delay.) The grouping file provides information on the relative placements of the primitive gates in the design. Because each design is a different size, ArtistII produced different size layouts. Being different in this manner, each of these layouts provides different layout information to the training set. The number of rows and columns for these layouts are also shown in Table 4. We wanted to keep the layouts as square as possible, with an equal or close to equal number of rows and columns when possible.

Table 4 Execution Times for ArtistII on Training Designs

Design Name	Number of Gates	Number of Transistors	Number of Rows	Number of Columns	Execution Time (CPU sec)
add4	43	146	8	8	840
add8	95	330	8	16	2278
add16	199	698	16	16	8278
add32	407	1434	32	16	18735

The fourth step is transforming data from the training designs and their layouts into a training set format. To make it easier to parse through the training design description, we used a tool called ClusterII to flatten the GLUE description. (The GLUE file can be hierarchical and very difficult to parse directly.) As described in Section 3.3.2.1, the primary function of ClusterII is to produce an initial gate clustering to improve the layouts. However, ClusterII can also produce a flattened file (.FLA) description of the design's GLUE file. We did not use ClusterII to improve the layouts because we wanted to model ArtistII strictly by itself. In addition, we manually checked the input GLUE files and the resultant flattened files to ensure that no optimization or other processing occurred.

A software tool called Match (written for this research) parses through each training design's flattened description file to get information on each of the primitive gates and the overall connectivity and topology of the design. This information allows the Match tool to generate a graphical representation of the design. The graphical representation is stored in an adjacency matrix. This matrix shows which gate pairs are connected. We can start at any gate and traverse through this matrix to get the topology of the design in relation to that gate. (We also use this matrix to get the critical path delay. This is explained later in Section 4.4.)

Information on each of the gates is stored in a record. This information includes the gate's name (identifier), function, number of external inputs (from the outside world), number of internal inputs (from within the design), height, width, area, intrinsic delay (for both rising and falling output transitions), and whether or not the gate is an input gate or output gate. An input gate has external inputs and an output gate has an output to the outside world. In addition, in the form of linked lists, this record holds information about the gates that are one and two levels away. This is the topology information described in the PDM file in Figure 14.

Each gate's height, width, area, and intrinsic delay come from a physical characteristics library. This library holds physical information for each primitive gate the layout tool can handle. The rest of the information comes from the design's flattened description file. For each training instance, the gate description, connectivity, and topology information provides values for the features describing each gate in the gate pair and for describing how the gates relate to each other. The Match tool parses

the .GRP file to obtain the relative placement information from the design's layout. This information provides the correct concept for each training instance.

Now that we have built a training set, we need to determine if it is complete or if it needs more or less information. This requires having the learning system train on it and produce a set of rules. This also requires using these rules on several test designs and measuring the performance. We did this for the training set using all 25,069 training instances and all features described by the PDM file in Figure 14. We used the model on the test designs shown in Table 5. (This table just shows the design name, function, and the number of gates in the design. The results are described in Section 5.2). It turns out the few rules based on the topology features had very low certainty factors (< 0.400), so we eliminated them. In addition, eliminating these features from the training set did not affect the model's performance. The individual gate features did not produce rules with high certainty factors either, so we eliminated these features. Once again, eliminating these features had no impact on the model's performance.

Table 5 Set of Test Designs

Design Name	Function	Number of Gates
add4	4-bit adder	43
add12	12-b adder	147
add24	24-b adder	303
abs16	16-b absolute value	208
inc4	4-b incrementer	23
inc8	8-b incrementer	51
inc16	16-b incrementer	107
sub4	4-b subtracter	48
sub8	8-b subtracter	104
sub16	16-b subtracter	216
addsub4	4-b adder/sub	64
addsub8	8-b adder/sub	132
addsub16	16-b adder/sub	268
mult2	2-b multiplier	40
mult4	4-b multiplier	163
mult6	6-b multiplier	389
mult8	8-b multiplier	711
mult16	16-b multiplier	2959

We were also able to reduce the number of training instances in the training set. Initially, we generated training instances for all possible gate pairs. We thought it would be important to learn how each gate related to each of the other gates. However, after applying the model to the test designs in Table 5, we found we only needed to know relative placement concepts for gate pairs that were logically connected. In the case of delay, the estimator calculates delay as a function of wirelength between two gates. If these two gates are not logically connected, there is no wire and no delay. The wirelength comes from the predicted relative placement concept for the two gates, therefore we do not need to know the relative placement of unconnected gates for calculating delay. In the case of area, we have the information we need from a list of all logically connected gate pairs. This is because this list must include each gate in the design in at least one gate pair. So each gate must have a relative placement concept associating it with at least one other gate in the estimated layout. By not generating training instances for gate pairs that were not logically connected, we reduced the size of the training set immensely.

For these reasons, we were able to reduce the PDM file shown in Figure 14 down to the PDM file shown in Figure 16. We were also able to reduce the number of training instances from 25,069 down to only 616. At this point, we were able to reinstate the training instances from the 32-bit adder. Because we now consider only logically connected gate pairs, the 32-bit adder provides only 528 training instances, instead of 82,621. Adding the 32-bit adder information increased the number of training instances in the training set to 1144.

The first line in the PDM file in Figure 16 shows we used the same relative placement concepts as in Figure 8. However, we eliminated the *log_conn* feature because we only consider logically connected gate pairs. The next two features, *com_dest* and *com_src* remained intact. For the other features, only the admissible values were changed. Instead of looking for the symbolic values of $<$, $=$, and $>$, we now look for $==$ (equal) or $!=$ (not equal). This made the features more general and less dependent on the order of gates in the gate pair. The last feature *name* was added to identify the gates in the gate pair. It has an admissible feature of *ignore* because it is only an identifier feature and we do not want it used in any rules. Now that we have explained how we scaled down to this reduced PDM file by eliminating

features and generalizing others, we can describe actual training instances.

```
near_s, near_d, close_1, close_2, far_1, far_2, way_out. | concepts
com_dest:       yes, no.
com_src:        yes, no.
a1_a2:          ==, !=.
h1_h2:          ==, !=.
w1_w2:          ==, !=.
nt1_nt2:        ==, !=.
nx1_nx2:        ==, !=.
ni1_ni2:        ==, !=.
fun1_fun2:      ==, !=.
name:           ignore.
```

Figure 16 Reduced Partial Domain Model

Figure 17 shows some training instances based on the reduced PDM file. These training instances come from the first training design, the 4-bit adder. Each training instance lists the values in the same order as the features in the PDM file. The next to last value is the gate pair identifier. The first number is the number of the first gate in the pair and the second number is the number of the second gate. The gate numbers are based on the order of gates in the flattened description file. The first gate described in this file is gate 0. These numbers are also used in the grouping file to indicate the gate's position in the layout. The last value in the training instance is the correct concept for that training instance. For example, the first training instance shows that gates 0 and 16 (the first two gates in the training design that are logically connected) do not have a common destination or common source. (The gate identifiers come from the order of gates in the training design's flattened description file. Gate 0 is the first gate listed and gate 16 is the 17th gate listed.) The remaining values in this training instance show the two gates are equal in area, height, width, number of total inputs, number of external inputs, and number of internal inputs. They also have the same function. The layout concept for these two gates is *close_2*. This indicates that in the actual layout for the 4-bit adder, gates 0 and 16 are four or five positions apart. The rest of the training set (not shown here due to its length) includes training instances from the 4-bit adder, 8-bit adder, 12-bit adder, 16-bit adder, and 32-bit adder. The Match tool produces a set of training instances for each

of these designs, one at a time. The training set is a concatenation of all the training instances.

```
no,no,==,==,==,==,==,==,==,0/16,close_2.
yes,no,==,!=,!=,!=,!=,==,!=,0/22,close_1.
no,no,==,!=,!=,!=,==,!=,!=,0/23,far_1.
no,no,!=,!=,!=,!=,!=,!=,!=,0/25,near_s.
no,no,==,==,==,==,==,==,==,1/23,far_1.
no,no,==,!=,!=,!=,==,!=,!=,1/24,near_s.
yes,no,==,==,==,==,!=,!=,!=,1/31,near_d.
no,no,==,==,==,==,==,==,==,1/32,close_2.
no,no,!=,!=,!=,!=,!=,==,!=,1/34,far_2.
no,no,==,==,==,==,==,==,==,2/32,far_1.
no,no,==,!=,!=,!=,==,!=,!=,2/33,close_1.
```

Figure 17 Example Training Instances

The fifth step is to build other training sets if we need to model the target layout tool using different configurations. For this example, we only wished to model ArtistII at 10,000 iterations so we do not need to build other training sets.

This section described the reasons behind a training set and how to actually build one. The next section describes how and why the learning system analyzes the example training set to produce a set of rules.

4.2 Analyzing the Training Set

The learning system produces a set of rules by learning about relationships between the features and the concepts in the training set. Therefore, the first step in analyzing the training set is determining if there are any relationships between the design features and the layout concepts. These relationships are represented by rules. The learning system finds these rules by performing a general-to-specific search of the rule space for the rules describing each concept. The rule space for each concept is structured as a tree. Each concept has its own tree. The root of this tree is the most general rule: every instance belongs to the concept. The tree branches out as the rules are specialized. The rules are specialized by either adding a conjunct to the

left-hand side or making an existing conjunct more specific. Restricting a numeric range or replacing a value with a more specific value makes conjuncts more specific.

Because we are using certainty factors as search parameters, the rules are made more specific until rules that satisfy the certainty factor threshold are found. These rules are considered acceptable rules and are added to the rule set. (Acceptable rules are those rules that satisfy the user-specified biases for the learning system.) When all acceptable rules have been found, the rules are evaluated by applying them to the training set. This determines the coverage and the performance. (This performance is really the apparent error rate as described in Section 2.3.) In this context, coverage means what percentage of the training instances match the left-hand side of one or more rules. A coverage of 100% means that each training instance fires at least one rule. The performance indicates how well the rules predict the correct concepts. A performance of 75% means three-fourths of the predicted concepts are correctly predicted.

For our example, we started out with a certainty factor threshold of 0.9. (The author of the Rule Learner system, F.J. Provost indicated that starting at 0.9 and decreasing by 0.1 would provide the best results for an implementation using certainty factors.) A threshold value of 0.9 means only those rules whose certainty factor equals or exceeds 0.9 are acceptable rules. However, no acceptable rules were found for a threshold of 0.9. We then lowered the threshold to 0.8. This time eight rules were found. However, these eight rules only covered 5.4% of the training instances and predicted correct concepts for only 4.5% of those covered. We wanted to have all (100%) training instances covered, so we lowered the threshold, this time to 0.7. The learning system found 21 acceptable rules that covered 25.2% of the training instances, with 12.8% correct. So we lowered the threshold to 0.6%. This produced 35 rules with 100% coverage and 33.0% of the training instances correctly predicted. For explanation purposes, Figure 18 shows five of the rules from this set.

The first rule says that if the two gates do not have a common destination and do have a different number of internal inputs, then their predicted concept is *near_s*. The statistics in this rule indicate 87% of the training instances with the concept of *near_s* satisfy these conditions (no common destination, different number of internal inputs). The statistics also indicate only 52% of the training instances with concepts

```
(com_dest no) (ni1_ni2 !=)  ==> near_s
 { pos=  0.87,  neg=  0.52,  cf=  0.623
  p=71, n=547, tp=82, tn=1062 }

(com_dest no) (fun1_fun2 !=)  ==> near_s
 { pos=  0.88,  neg=  0.57,  cf=  0.603
  p=72, n=603, tp=82, tn=1062 }

(h1_h2 !=) (ni1_ni2 !=)  ==> close_1
 { pos=  0.86,  neg=  0.56,  cf=  0.602
  p=106, n=574, tp=123, tn=1021 }

(com_dest no) (nx1_nx2 !=) (fun1_fun2 !=)  ==> close_1
 { pos=  0.51,  neg=  0.33,  cf=  0.602
  p=63, n=339, tp=123, tn=1021 }

(com_dest yes) (a1_a2 !=)  ==> close_2
 { pos=  0.15,  neg=  0.05,  cf=  0.735
  p=19, n=48, tp=130, tn=1014 }

(nx1_nx2 ==) (fun1_fun2 !=)  ==> far_1
 { pos=  0.44,  neg=  0.28,  cf=  0.603
  p=52, n=288, tp=119, tn=1025 }

(com_dest yes) (nx1_nx2 !=)  ==> way_out
 { pos=  0.10,  neg=  0.03,  cf=  0.791
  p=58, n=14, tp=585, tn=559 }
```

Figure 18 Example Set of Rules No. 2

other than *near_s* satisfy these conditions. So this rule has a .623 certainty factor according to Equation 3-2. It should be noted here that this certainty factor is not a probability of the rule's truth. It is a value that indicates the learning system's belief in the rule (based on the training set) and should only be used to rank order this rule with other rules indicating the same concept.

The second step in analyzing the training set is taken if no relationships are found. We would have to adjust some of the learning system's biases and try analyzing the training set again. In the example case, we used a beam width of 50 for the beam search strategy. We also allowed 6 conjuncts in the left-hand side of the rules. This

means that up to 6 features (two-thirds of the possible features) could be listed as conditions. These bias settings produced 35 acceptable rules. If no rules were produced, we could have used a wider beam width and/or allowed more conjuncts. Then, if the learning system still produced no rules, we would have to build and use another training set.

The third step is building a set of rules (the model) describing the relationships that were found. The learning system does this automatically. The set of rules we obtained for our example is a model of ArtistII at 10,000 iterations for different size adder designs. Applying these rules to other adder designs is explained in the next section.

4.3 Applying the Model

In this section, we explain in detail how we use the layout tool model to obtain layout information. We applied the model to five test designs to illustrate how to apply the model. (In the next two sections, we use these same five designs to illustrate how to evaluate the performance of the model.) Test designs are treated the same as target designs, except they have associated layout information obtained from actual layouts as a reference. The test designs we used from Table 5 are the 12-bit adder, 24-bit adder, 16-bit absolute value operator, and 4-bit multiplier. (We included the 4-bit multiplier to test the hypothesis that rules learned from adders are applicable to multipliers as well.)

The main idea behind the layout tool model is that it represents the target layout tool in a form a high-level synthesis system can use to get area and delay estimates. The synthesis system applies the model to a target design to get layout information from which the system can derive area and delay estimates. To do this, the model is based on how the target layout tool places logically connected gate pairs relative to each other. When applied to a target design, the model produces a list of predicted relative placements for the logically connected gates. As described in Section 4.5 and Section 4.4, this list provides placement information for estimating area and wirelength information for estimating delay.

Now that we have explained the theory behind applying the model, we can illustrate with actual examples. The first step in applying the model is pre-processing the test designs into a format so we can apply the rules. This format is the same as the training set format, but without the correct concepts. Therefore, we have to first transform the test designs from their VHDL descriptions into GLUE format descriptions. Then we use ClusterII to get the flattened file description. (Once again, ClusterII does not change anything while it is flattening the GLUE file description.) Pre-processing the designs in this way keeps our method consistent because the Match tool can treat test designs the same as training designs.

However, because these are test designs, we used ArtistII to produce reference layouts for each one. We used the same number of iterations of ArtistII (10,000) as we did for the training designs. Table 6 shows the five test designs and the number of rows and columns in the actual layout for each one. We tried to keep the layout as square as possible with an equal or close to equal number of rows and columns. We relaxed this restriction for the multiplier to see if it affected the estimates and found out that it had no effect.

Table 6 Test Design Layout Information

Design Name	Number of Gates	Number of Transistors	Number of Rows	Number of Columns
add4	43	146	8	8
add12	147	514	16	16
add24	303	1066	16	32
abs16	208	638	16	16
mult4	163	550	8	32

We used Magic to obtain the actual height and width of each layout, and IRSIM to obtain the worst-case critical path delay. This information will be used to evaluate the performance of both the learning system and the area and delay estimator.

The Match tool processes the flattened descriptions for each of the test designs in the same manner as the training designs. The output is a test data file that looks exactly like a training set, but without the concept information. Figure 19 shows a sample set of data from the 4-bit multiplier. The Match tool also produces a file that describes the physical limitations for the test design's estimated layout. This is the

```
no,no,==,==,==,==,==,==,==,0/114.
no,no,==,==,==,==,==,==,==,1/107.
yes,no,==,!=,!=,!=,==,!=,!=,8/11.
no,yes,!=,!=,!=,!=,==,!=,!=,8/137.
no,no,==,==,==,==,==,==,==,8/152.
no,no,==,==,==,==,==,==,==,9/11.
no,no,==,!=,!=,!=,==,!=,!=,9/158.
no,no,==,!=,!=,!=,==,!=,!=,10/12.
no,no,==,==,==,==,==,==,==,10/115.
no,no,!=,!=,!=,!=,==,!=,!=,11/137.
no,no,==,!=,!=,!=,==,!=,!=,12/13.
no,no,==,!=,!=,!=,!=,!=,!=,141/159.
no,no,==,!=,!=,!=,!=,!=,!=,142/143.
```

Figure 19 Example Data from 4-Bit Multiplier

.STAT file shown on the right hand side of Figure 15. The Match tool produces the .STAT file by first asking the user how many rows to use in the estimated layout. (In the case of the test designs, we used the same number of rows as in the actual layouts to provide a valid comparison.) The Match tool uses this number of rows and the number of gates in the test design to determine the required number of columns. The number of rows and number of columns determines the size of the estimated layout and the physical limitations. The physical limitations are based on the fact there can only be so many gate pairs that are next to each other on the same row, or next to each other on different rows, or so many positions apart. The physical limitations file indicates how many of each layout concept is possible in the estimated layout. This physical limitations file is used in the next step.

The second step is applying the model to each of the test design's data files to obtain predicted layout information. As described in Section 3.3.3.1, one of the routines in the learning system is called the *Inference Engine*. This routine applies a set of rules to a data file and produces a list of predicted concepts. The list of predicted concepts includes each gate pair and their predicted relative placement concept.

For this research and this example, the Inference Engine applies the rules as follows. First, a list of all concepts is created for each gate pair. For our example,

each gate pair has a list of the seven concepts shown in the PDM file in Figure 16. Initially, each concept is assigned a certainty factor of 0.0. The Inference Engine takes the first rule from the set of rules and begins applying it to the data instances for each gate pair. If a rule fires for a gate pair, the rule's certainty factor is combined with the predicted concept's current certainty factor for the gate pair according to Equation 4-1, where X and Y are the certainty factors to be combined. (This is the same as Equation 3-3. It is repeated here for easy reference.) This combined certainty factor becomes the current certainty factor for that concept.

$$cf_{comb}(X,Y) = X + Y(1 - X) \tag{4-1}$$

To illustrate, we will apply a couple of rules from Figure 18 to some of the gate pairs from Figure 19 and show how the certainty factors combine. The first rule in Figure 18 predicts a concept of *near_s* for any gate pairs that do not have a common destination and have an unequal number of internal inputs. This rule's certainty factor is 0.623. From the data in Figure 19, the first gate pair to fire the rule is 9/158. The current certainty factor for the concept *near_s* in this gate pair's list of concepts is combined with the rule's certainty factor. The current certainty factor is 0.0 and the rule's certainty factor is 0.623, so the combined certainty factor is 0.623. After combining the certainty factors, the list of concepts for the gate pair 9/158 looks like Figure 20. This rule is applied to the rest of the gate pairs and if any of them fire the rule, their concept lists will be updated in the same way.

```
Gate Pair:  9/158
near_s:  0.623
near_d:  0.0
close_1: 0.0
close_2: 0.0
far_1:   0.0
far_2:   0.0
way_out: 0.0
```

Figure 20 Concept List No. 1

The Inference Engine then takes the second rule and applies it to all gate pairs. The second rule from Figure 18 predicts a concept of *near_s* for any gate pairs that do not have a common destination and do not have the same function. Gate pair 9/158 is one of the pairs that fires this rule so its concept list is updated. The rule's certainty factor of 0.603 is combined with the current certainty factor of 0.623 for the *near_s* concept. This produces a combined certainty factor of 0.850 according to Equation 4-1. Now the concept list for gate pair looks like Figure 21.

```
Gate Pair:  9/158
near_s:  0.850
near_d:  0.0
close_1: 0.0
close_2: 0.0
far_1:   0.0
far_2:   0.0
way_out: 0.0
```

Figure 21 Concept List No. 2

After applying the second rule to the remaining gate pairs and updating their concept lists as necessary, the Inference Engine takes the third rule and applies it. This rule predicts a concept of *close_1* with a certainty factor of 0.602 for any gate pair whose gates have different heights and a different number of internal inputs. Gate pair 9/158 is one of the gate pairs that fires this rule, so its concept list is updated. The current certainty factor for the *close_1* concept is 0.0, so combining it with the rule's certainty factor produces a new certainty factor of 0.602. The concept list for gate pair 9/158 now looks like Figure 22.

This rule application process is continued until all rules have been applied to all gate pairs. When this process is finished, each gate pair will have a list of predicted concepts with their final certainty factors. Each gate pair's list of concepts is then rank ordered by certainty factor in descending order. The concept with the highest certainty factor becomes the first choice for the gate pair. The concept with the next highest certainty factor becomes the second choice, and so on. The final concept list, rank ordered by certainty factor, for gate pair 9/158 is shown in Figure 23. This

```
Gate Pair:  9/158
near_s:  0.850
near_d:  0.0
close_1: 0.602
close_2: 0.0
far_1:   0.0
far_2:   0.0
way_out: 0.0
```

Figure 22 Concept List No. 3

list indicates that for gate pair 9/158, the concept *near_s* is the first choice, with the concept *far_1* a very close second. In fact, either concept could be used as the first choice because they are so close. In ties like this, the concept representing the smallest distance between the two gates is listed first, because the cost function of the target layout tool is based on finding the minimal area.

```
Gate Pair:  9/158
near_s:  0.991
far_1:   0.990
close_1: 0.975
way_out: 0.605
near_d:  0.0
close_2: 0.0
far_2:   0.0
```

Figure 23 Final Concept List

At first glance, the second choice of *far_1* after the first choice of *near_s* does not make sense spatially. However, as it turns out, the set of rules does not contain any rules predicting the concept of *near_d*. This is because for this particular training set, there is not enough evidence supporting any rules for the concept of *near_d*. In addition, based on evidence in the training set, the gates in a gate pair with features like gate pair 9/158 want to be placed next to each other, but if that is not possible, a placement further away is warranted. Otherwise, if we were to always use the next spatial concept as a second choice, we would violate what was learned from the

training set. This, in turn, would go against the goal of this research: using machine learning to model the input-to-output relationships from using layout tools.

Now that we have rank-ordered each gate pair's concept list, the entire list of logically connected gate pairs is rank ordered in descending order using the certainty factor from the gate pair's first choice concept. The gate pair with the highest certainty factor for the first choice concept is now the first gate pair in the list. The gate pair with the next highest certainty factor for the first choice is the second in the list, and so on.

Starting at the top of this list, each gate pair is assigned its first choice concept, unless that concept is unavailable due to physical limitations. (This means it is not possible to place the two gates the appropriate number of positions apart.) Each time a concept is assigned, the physical limitation file is updated. When that concept is no longer possible for the remaining gate pairs in the predicted layout, any unassigned gate pairs in the list having that concept as their first choice must now use their second choice, if it is available. This is another example of a gray area in the method. It is not necessary to assign a gate pair its first choice of relative placement concepts. For example, if the physical limitations allow only so many concepts of *near_s*, the gate pairs with the highest certainty factors for this concept should have it assigned first. When the available number of this concept runs out, then the gate pair's second choice can be assigned. Having a gray area gives the method flexibility when assigning concepts so physical limitations of the domain can be satisfied.

The assignment process is repeated until all gate pairs have an assigned concept. The end result is a file that lists all gate pairs with their predicted concepts. This file is rank ordered so the gate pairs near the top have been assigned concepts with the best certainty factors. The gate pairs towards the end of the list may not have been assigned their first or second choice concepts, but the certainty factors are lower towards the end due to the rank ordering. This means there is a higher probability that the first choice concepts may be incorrect as it is. The physical limitations define the bounds of the predicted layout and enhances the overall method.

If none of the rules fire for any of the gate pairs, the third step is taken and a different layout tool model is used. In our example, we obtained a complete set of

predicted concepts for each of the test designs. We did not need to use a different model. Now that we have the predicted layout information, we are ready to move on to estimating area and delay.

4.4 Estimating Delay

One of the performance measures for a digital design is the execution speed. Designers and users need to know how fast a design can process the inputs and produce a usable output. In following the old adage, "A chain is as strong as its weakest link", we can say a design is as fast as its slowest link. This means the slowest path from input to output determines how fast the design can execute. This slowest path is called the *critical path.* The delay of this *critical path* is inversely proportional to its speed. The lower the delay, the faster the speed, and vice versa.

The overall delay of the critical path is a sum of all the delays between nodes on the path. For this research, we calculate the delays between nodes as a function of the driving node's intrinsic delay, the delay due to the length of wire on the driving node's output, and the load that the driven node(s) place on the driving node. The driving node's intrinsic delay comes from a physical library file. The load from the driven node(s) is a function of the number of driven nodes. This comes from the design's netlist. The length of wire on the driving node's output comes from knowledge of the design's layout. In a synthesis system using the method from this research, the system does not know about the design's actual layout. However, it does have layout information from applying the layout tool model. This layout information is in the form of predicted relative placement concepts for all logically connected gate pairs. The relative placement of two gates translates directly into the length of wire needed to connect them.

Therefore, the first step in estimating delay is obtaining the wirelengths between all logically connected gates. The relationship between relative placement concepts and wirelength is shown in Table 7. For example, a relative placement concept of *close_1* predicts the two gates as being either two or three positions apart. This translates into a wirelength of about 210 microns. These wirelength relationships were determined by performing a statistical analysis of several actual layouts. We

found the average wirelength between two gates next to each other to be about 70 microns. The wirelengths for gates placed further apart is nearly linear. Because we are interested in worst case delay, we used the largest number of positions apart in each concept to determine the wirelength.

Table 7 Relationship of Wirelength to Relative Placement

Relative Placement Concept	Number of Positions Apart	Approximate Wirelength (microns)
near_s	1	70
near_d	1	70
close_1	2-3	210
close_2	4-5	350
far_1	6-7	490
far_2	8-9	630
way_out	10+	>700

Here again, the concept of *way_out* has proven sufficient for gate pairs that are further than 10 positions apart. This concept is only a categorization of these gate pairs. The predicted relative placement gives us the number of positions apart and the estimator calculates the necessary wirelength. For example, a gate pair that is 20 positions apart would have the concept of *way_out*, but the wirelength between the two gates would be 1400 microns (20×70).

To obtain the wirelength information, the Match tool reads in the list of gate pairs and their relative placement concepts. The Match tool then translates each relative placement concept into a wirelength value for the gate pair according to the relationships in Table 7. These wirelength values are used in the next step to determine the delay between the gates on the critical path.

The second step in estimating delay is performing a critical path analysis. This analysis is first performed based on the critical path having a low-high transition on the path's output. The design is treated as a forest of trees. Each output gate is a root for one of these trees. The input gates are the leaves. The estimator starts at an output gate and travels down the tree in a depth-first search until it reaches an input gate. The estimator then backtracks up the tree one level, to its parent gate.

When the estimator backtracks, it calculates and stores the delay from the child gate to the parent gate. (This delay includes the gate's intrinsic delay from the physical characteristics library and the delay due to the wires and interconnect loading.) The estimator then travels down the next branch, if there is one. When an input gate is reached, the estimator backtracks and calculates the delay. This is repeated for all branches from the parent gate. The branch with the worse delay becomes part of the critical path for the root output gate. The worse case delay becomes the delay so far for the parent gate. The estimator then backtracks up to the next level, the parent of the parent, and calculates the delay. Once again, the estimator travels down each of the other branches until it reaches the input gates, backtracks and calculates the delays. In a depth-first fashion, the estimator searches the entire tree, calculating the delays and keeping track of the critical path as it backtracks. Eventually, the estimator will backtrack all the way back to the output gate. When it reaches the output gate, both the critical path and the critical path delay for that output will be known. This process is repeated for each tree. After all trees have been searched and all critical paths and delays are known, the critical path with the longest delay becomes the critical path for the test design.

As the estimator travels down each tree, it assigns alternating values of 1 and 0 to the gates. In this case, the estimator is calculating the critical path delay for the output gate having a low-high transition. So the estimator assigns the output gate a value of 1. The next gate down the tree is assigned a value of 0. This is repeated until every gate in the tree has an assigned value. The value assigned to a gate determines what delay equation is used for that gate. If a value of 1 is assigned, Equation 4-2 is used to calculate the delay. If a value of 0 is assigned, Equation 4-3 is used. This method makes the critical path delay appear data dependent. Otherwise, using the same delay equation for each gate produces a very pessimistic worse case delay.

The delay equations come from research on the TinkerTool [65] system. Equation 4-2 calculates the delay for a gate having a low-high transition. Equation 4-3 calculates the delay for a gate having a high-low transition. These equations use the average wire length, AWL, and the number of interconnects, INC, to determine the delays. The average wire length determines the delay due to all of the wire connected to the output of the gate, and the number of interconnects determines the delay due

to the load on the output (i.e., the number of gates being driven by this gate).

$$delay_{0-1} = AWL \times 0.0022 + INC \times 0.3740 \tag{4-2}$$

$$delay_{1-0} = AWL \times 0.0010 + INC \times 0.1217 \tag{4-3}$$

Using these equations and the predicted layout information, the estimator can find the critical path. Figure 24 shows the estimated critical path for the 4-bit multiplier test design with a low-high transition on the path's output. Each line shows the delay information for one gate on the path. The gate's identifier begins the line. Next comes the transition information. A "0" indicates a high-low transition and a "1" indicates a low-high transition. The value immediately following the "" symbol indicates the time at which the transition is complete. The next value (in parentheses) indicates the elasped time for the transition from beginning to end.

Starting at the bottom of the figure, the last line shows the input gate on the critical path. For example, this input gate is named *_10* in the 4-bit multiplier design. The input is applied to this gate at time 0.00. This gate goes low at time 1.01. This is 1.01 ns after the input is applied. The next gate, *x1y1*, goes high at time 6.05. This is 5.05 ns after gate *_10* goes low. This indicates a delay of 5.05 ns. The critical path can be traced by starting at the bottom and working up to the top line. The gate at the top is the output gate for the critical path. For this design, following the critical path up to the output gate indicates a overall delay of 52.64 ns.

The third step in estimating delay is repeating the second step by performing a critical path analysis based on the critical path having a high-low transition on the path's output. Performing two critical path analyses gave us more flexibility when comparing to critical path delays from the IRSIM simulator. Because the actual delays come from running simulations on the test designs using a set of input vectors, the critical path delay is input data dependent. In addition, the output gate on the critical path does not always have a high or low transition. We needed a way to choose the appropriate output transition so we could make valid comparisons and calculate the right errors. Figure 25 shows the estimated critical path for the 4-bit

```
product[7] -> 1 @ 52.64  (0.40ns)
_buf_1.t[0] -> 0 @ 52.24  (0.75ns)
c4c2 -> 1 @ 51.49  (2.58ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_0._4 -> 0 @ 48.91  (1.12ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_0.g -> 1 @ 47.79  (3.32ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_0._1 -> 0 @ 44.47  (1.28ns)
c4c1 -> 1 @ 43.19  (3.33ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_1._4 -> 0 @ 39.86  (0.97ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_1.g -> 1 @ 38.89  (3.00ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_1._1 -> 0 @ 35.90 (1.20ns)
c4c0 -> 1 @ 34.69 (2.91ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_2._6 -> 0 @ 31.79 (0.94ns)
c3c0 -> 1 @ 30.84  (2.84ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_6._4 -> 0 @ 28.00  (1.19ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_6.g -> 1 @ 26.81  (3.48ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_6._0 -> 0 @ 23.33  (1.42ns)
s2s1 -> 1 @ 21.90  (3.02ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_7._xor_15.t[0] -> 0 @ 18.89 (0.70ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_7._5 -> 1 @ 18.19  (2.24ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_7._xor_14.t[0] -> 0 @ 15.95 (0.55ns)
c1c1 -> 1 @ 15.40  (2.91ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_10._4 -> 0 @ 12.50  (0.82ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_10.g -> 1 @ 11.67  (4.13ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_10._1 -> 0 @ 7.54  (1.48ns)
x1y1 -> 1 @ 6.05  (5.05ns)
_10 -> 0 @ 1.01  (1.01ns)
```

Figure 24 Estimated Critical Path for 4-Bit Multiplier (Output High)

multiplier with a high-low transition on the path's output. The overall delay for this critical path is 44.99 ns.

For comparison purposes, Figure 26 shows the critical path that IRSIM found for the 4-bit multiplier by running a simulation using electrical information from the actual layout. This simulation showed the real critical path has the output going low, with an overall delay of 44.5 ns. If we compare Figure 26 to Figure 25 line by line, we can see the delay estimator tracks well with the simulation from IRSIM. (Note: the IRSIM tool produces its critical path list in reverse order from the estimated critical path lists. However, each line contains the same type of data found in each line of the estimator's critical path descriptions.)

```
product[7] -> 0 @ 44.99  (0.30ns)
_buf_1.t[0] = 44.69  (1.51ns)
c4c2 -> 0 @ 43.19  (1.09ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_0._4 = 42.09  (2.32ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_0.g -> 0 @ 39.77  (0.92ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_0._1 = 38.86  (2.78ns)
c4c1 -> 0 @ 36.08  (1.33ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_1._4 = 34.75  (2.00ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_1.g -> 0 @ 32.75  (0.77ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_1._1 = 31.98  (2.61ns)
c4c0 -> 0 @ 29.37  (1.29ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_2._6 = 28.08  (2.26ns)
c3c0 -> 0 @ 25.82  (1.11ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_6._4 = 24.71  (2.48ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_6.g -> 0 @ 22.22  (0.99ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_6._0 = 21.23  (3.10ns)
s2s1 -> 0 @ 18.13  (0.87ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_7._xor_15.t[0] = 17.26  (2.83ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_7._5 -> 0 @ 14.43  (0.56ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_7._xor_14.t[0] = 13.87  (2.51ns)
c1c1 -> 0 @ 11.36  (1.29ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_10._4 = 10.07  (1.67ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_10.g -> 0 @ 8.40  (1.28ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_10._1 = 7.12  (3.13ns)
x1y1 -> 0 @ 3.99  (1.60ns)
_10 = 2.38  (2.38ns)
```

Figure 25 Estimated Critical Path for 4-Bit Multiplier (Output Low)

Table 8 shows the delay estimates we obtained for the five test designs and the percent error between these delay estimates and the actual delays from IRSIM simulations. This table shows the model performs quite well estimating delay in its final implementation for this example. This table also shows that even though the model is based on a training set using all adder designs, the model works well on other types of adder-based designs. The percent errors for the five test designs are all under 10%. (More results are presented in Chapter 5.)

```
y_1 -> 1 @ 5000.0ns , node was an input
_10 -> 0 @ 5000.9ns    (0.9ns)
x1y1 -> 1 @ 5005.9ns    (5.0ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_10._1 -> 0 @ 5007.3ns  (1.4ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_10.g -> 1 @ 5009.3ns  (2.0ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_10._4 -> 0 @ 5010.4ns  (1.1ns)
c1c1 -> 1 @ 5012.5ns  (2.1ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_7._xor_14.t_0 -> 0 @ 5013.5ns (1.0ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_7._5 -> 1 @ 5017.2ns  (3.7ns)
s2s1 -> 0 @ 5020.4ns  (3.2ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_6._0 -> 1 @ 5022.0ns  (1.6ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_6.p -> 0 @ 5023.2ns  (1.2ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_6._6 -> 1 @ 5024.9ns  (1.7ns)
c3c0 -> 0 @ 5027.2ns  (2.3ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_2._6 -> 1 @ 5029.4ns  (2.2ns)
c4c0 -> 0 @ 5031.0ns  (1.6ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_1._1 -> 1 @ 5032.4ns  (1.4ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_1.p -> 0 @ 5033.4ns  (1.0ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_1._6 -> 1 @ 5034.7ns  (1.3ns)
c4c1 -> 0 @ 5037.0ns  (2.3ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_0._1 -> 1 @ 5038.5ns  (1.5ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_0.p -> 0 @ 5039.4ns  (0.9ns)
addc_b1_d0_c1.addc_b1_d0_c1_s_0._6 -> 1 @ 5041.2ns  (1.8ns)
c4c2 -> 0 @ 5042.3ns  (1.1ns)
_buf_1.t_0 -> 1 @ 5043.9ns  (1.6ns)
product_7 -> 0 @ 5044.5ns  (0.6ns)
```

Figure 26 IRSIM Critical Path for 4-Bit Multiplier

4.5 Estimating Area

To keep the estimation process consistent, the area estimates are based on the same information as the delay estimates. This information is the list of gate pairs and their relative placement concepts produced by applying the model to a target design. The relative placement concepts indicate where the two gates are placed in relation to each other and is used to generate an estimated layout. This estimated layout is not a real layout nor should it be misconstrued as a replacement for a real layout. It only provides information on the predicted placement of gates so the height and width can be estimated.

Table 8 Delay Estimates for Example Test Designs

Design Name	Number of Gates	Actual Delay (ns)	Estimated Delay (ns)	Percent Error
add4	43	19.1	19.2	0.54%
add12	147	61.5	60.35	-1.87%
add24	303	134.0	125.06	-6.67%
abs16	208	109.0	112.36	3.08%
mult4	163	44.5	44.99	1.11%

Therefore, the first step in estimating area is using the list of relative placement concepts to produce an estimated layout. As described in Section 3.3.3.3, the estimator (part of the Match tool) "grows" an estimated layout. The first gate from the first gate pair in the list of predicted concepts is planted in the center of the estimated layout. This becomes the "seed" gate. The second gate from the first gate pair is placed in relation to the seed gate according to the predicted concept for the gate pair. In other words, when the estimated layout is finished, this gate will be located the appropriate Manhattan distance away from the seed gate. Due to the rank ordering of the gate pairs according to certainty factors, the first gate pair in the list has an assigned concept with the best certainty factor of all. This means this gate pair has the best chance of having the correct relative placement concept. This is why we start at the top of the list.

The placement of these two gates is shown in Figure 27 for the 4-bit adder. The first gate pair in the list consists of gates 2 and 40. Gate 2, the seed gate, is located in the middle of the estimated layout and gate 40 is located four positions away in accordance with its predicted relative placement concept. (The nils are place holders.) Therefore, gate 40 has a Manhattan distance of four positions away from gate 2.

Because ArtistII layouts require the number of rows and number of columns to be a power of two, there will always be an even number of rows and columns. This means there really is no exact "middle" position. So the estimator first divides the estimated layout into an upper half and a lower half. For this example, the upper half consists of the first four rows and the bottom half consists of the last four rows. The estimator then divides the bottom half into two quarters. The left quarter consists of the first four columns, and the right quarter consists of the last four columns.

```
(add4.est
(row r1 nil nil nil nil nil nil nil nil)
(row r2 nil nil nil nil nil nil nil nil)
(row r3 nil nil nil nil nil nil nil nil)
(row r4 nil nil nil nil nil nil nil g40)
(row r5 nil nil nil nil g2  nil nil nil)
(row r6 nil nil nil nil nil nil nil nil)
(row r7 nil nil nil nil nil nil nil nil)
(row r8 nil nil nil nil nil nil nil nil)
(column a1 r1 r2 r3 r4 r5 r6 r7 r8)
)
```

Figure 27 Estimated Layout for 4-Bit Adder No. 1

The seed gate is planted in the upper left position of the lower right quarter of the estimated layout. The seed gate could also be planted in the upper right position of the lower left quarter, or the lower right of the upper left quarter, or even in the lower left of the upper right quarter. Any of these four locations are close to the middle of the estimated layout, so it does not matter.

Once the seed gate is planted, the estimator starts looking for a location for the next gate at the lower right corner of the upper half of the estimated layout. In this case, this position satisfied the relative placement concept so gate 40 was placed there. If this location did not satisfy the concept, the estimator would scan to the left and evaluate each vacant (nil) position. If no acceptable positions were found in that row, the estimator would move up one row and start over from the right.

After the seed gate and its paired gate are placed, the list is searched in descending order for other gates connected to the seed gate. These gates are placed according to their predicted concepts. If a gate cannot be placed in a position that satisfies its relative placement concept, it is placed in the next best position. For example, a gate pair with a predicted concept of *close_1* means the two gates should be placed a Manhattan distance of four or five positions apart. Because one of the gates will already be located somewhere in the layout, the estimator only has to find an empty position for the other gate. This other gate should be located four or five positions away. If there are no empty positions that satisfy this requirement, the requirement

is relaxed and the gate may be located either three or six positions away. This placement method tries to first use what was learned from the training set, but if this is not possible, it then uses the physical limitations of the layout as a guide. This is another example of a gray area in the method. A gate may be placed close enough to where it needs to go to satisfy both the model and the physical limitations.

In the example, the next gate pair in the list that includes gate 2 includes gate 41. The relative placement concept for this gate pair indicates gate 41 should be placed two or three positions away from gate 2. This is shown in Figure 28.

```
(add4.est
(row r1 nil nil nil nil nil nil nil nil)
(row r2 nil nil nil nil nil nil nil nil)
(row r3 nil nil nil g41 nil nil nil nil)
(row r4 nil nil nil nil nil nil nil g40)
(row r5 nil nil nil nil g2  nil nil nil)
(row r6 nil nil nil nil nil nil nil nil)
(row r7 nil nil nil nil nil nil nil nil)
(row r8 nil nil nil nil nil nil nil nil)
(column a1 r1 r2 r3 r4 r5 r6 r7 r8)
)
```

Figure 28 Estimated Layout for 4-Bit Adder No. 2

Once all gates logically connected to the seed gate have been placed, the estimator goes back to the top of the list and finds the next unplaced gate. Because this gate does not share a predicted relative placement concept with the seed gate, it can be placed anywhere and not violate any rules. Therefore, it is placed in the next available position near the middle of the estimated layout. In the 4-bit adder example, gate 2 is only connected to gates 32, 33, 40, and 41. The estimator goes down the list and places each of these gates. Then the estimator goes back to the top and looks for the next unplaced gate. This is gate 1. It is not connected to gate 2, but it is placed next to gate 2. The end result of placing these gates is shown in Figure 29.

The placement process is repeated until all gates have been placed. Figure 30 shows the final estimated layout after all gates have been placed. Once again, this

```
(add4.est
(row r1 nil nil nil nil nil nil nil nil)
(row r2 nil nil nil nil nil nil g33 nil)
(row r3 g32 nil nil g41 nil nil nil nil)
(row r4 nil nil nil nil nil nil nil g40)
(row r5 nil nil nil nil  g2  g1 nil nil)
(row r6 nil nil nil nil nil nil nil nil)
(row r7 nil nil nil nil nil nil nil nil)
(row r8 nil nil nil nil nil nil nil nil)
(column a1 r1 r2 r3 r4 r5 r6 r7 r8)
)
```

Figure 29 Estimated Layout for 4-Bit Adder No. 3

estimated layout is only a vehicle to estimate area based on relative placement concepts. It should not be treated as a real layout or as a replacement for one.

```
(add4.est
(row r1 nil nil g37 g28 nil nil nil nil)
(row r2 nil g16 g11 g38 g29 g7 g33 g13)
(row r3 g32 g12 g24 g41 g5 g9 g25 g42)
(row r4 g23 g19 g34 g20 g17 g22 g31 g40)
(row r5 nil nil nil nil g2 g1 g14 g0)
(row r6 nil nil nil nil g4 g8 g6 g3)
(row r7 nil nil nil nil g30 g21 g39 g15)
(row r8 nil nil nil nil g35 g27 g26 g18)
(column a1 r1 r2 r3 r4 r5 r6 r7 r8)
)
```

Figure 30 Final Estimated Layout for 4-Bit Adder

Other layout "growing" techniques were tried, like starting in one corner and growing inward. We also tried starting in the left-most position in the middle row and alternating the growing cycle between the lower half of the layout and the upper half. The technique we presented here gave the best estimates. This is because every row has at least one gate, and there is at least one row that is completely filled with gates. This gives us height and width estimates slightly on the pessimistic side. A comparison of these estimates and the actual values is presented later in this section.

In addition, it turns out that placing groups of connected gates into and around other groups has a very small effect on the final area estimate. In other words, growing clusters of connected gates on top of clusters of other connected gates provided good results. This is because we are interested in the relative placement of logically connected gate pairs. If two gates are not logically connected, we can place them anywhere in relation to each other, and not violate any rules. In fact, in some cases, doing this may satisfy some physical limitation. One factor that allows this is that in most designs, all gates are indirectly connected to each other. This means that any given cluster usually has at least one interconnection to another cluster. This interconnection means there is a predicted relative placement for the two gates involved, one from one cluster and one from the other cluster. This relative placement helps to determine how the clusters are "grown" on top of each other.

An alternative to this is to predict a relative placement for all gate pairs. Then we would know where each gate should be placed according to other gates already placed. However, as described in Section 3.3.2.1, the total number of gate pairs for a design is of $O(n^2)$, where n is the number of gates. This would make it very difficult to handle large designs, even with only a few thousand gates.

Now that we have generated an estimated layout, we can derive an area estimate. There are two components to area estimation: the area of the gates themselves and the wire area. The estimator finds the area of the estimated layout by finding the area of the gates first and then adds in the required wire area.

Therefore, the second step in estimating area is finding the minimum height and width of the estimated layout due to just the gates themselves. First, the estimator finds the tallest gate in each row. The height of this gate becomes the height of that row. For example, in the fourth row in Figure 30, the tallest gate is gate 34. This gate has a height of 73 lambda. Therefore, the height of the entire row must be at least 73 lambda in order to accomodate this gate. The heights of the other gates do not change. They are shorter so they can fit in this row with no problem. Once the minimum height of each row is found, the heights of all the rows are summed to produce a minimum height of the estimated layout. The minimum height of the estimated layout in Figure 30 is 478 lambda.

Next, the estimator calculates the width of each row. In the example in Figure 30, the third row is the widest. It has a width of 298 lambda. This becomes the minimum width of the estimated layout.

The third step is adding in the wire area. For this research, we used a very simple model of wiring area. The more columns in a layout, the more horizontal wires needed. The more rows in a layout, the more vertical wires needed. The horizontal wires increase the height of the layout and the vertical wires increase the width. In addition, this model takes into account that densely populated layouts require extra area for wires than sparsely populated layouts. The wires have more room to travel among the gates in a sparsely populated layout.

Other wire area models like Rent's rule [21] and the Left Edge Algorithm [59] do not work well for the type of estimated layout used here. This estimated layout is based solely on the relative placement of logically connected gate pairs. It does not try to minimize a cost function based on interconnections or wire area. Using a wire area model like Rent's rule or the Left Edge Algorithm produced overly pessimistic wire areas.

The wire area model we used is a function of the number of rows, columns, and gates in the design. For small designs with less than 100 gates, the wire area did not contribute to the overall area. The estimated area of these designs is based solely on the minimum height and width due to the gates. For designs between 100 and 2000 gates, we used Equation 4-4 to increase the height and Equation 4-5 to increase the width.

$$ht\_fact = NUM\_COLS \times 2/100.0 + NUM\_NODES/10000.0 \qquad (4\text{-}4)$$

$$wd\_fact = NUM\_ROWS \times 4/100.0 + NUM\_NODES/10000.0 \qquad (4\text{-}5)$$

Equation 4-4 is based on the premise that more columns require more horizontal wires, and horizontal wires increase the height. It is also based on the premise that a dense layout (more nodes) requires more area for wires than a sparse layout. This is reflected in the number of nodes factor. Equation 4-5 is based on the premise that

more rows require more vertical wires, and vertical wires increase the width. This equation also takes into account the number of nodes.

For designs with more than 2000 gates where the number of columns was greater than or equal to the number of rows, we used Equation 4-6 to increase the height and Equation 4-7 to increase the width. The height factor did not change in this case (Equation 4-6 is the same as Equation 4-4). However, as the size of the design increased and because there are more columns than rows, the width factor decreased. This is reflected in Equation 4-7.

$$ht\_fact = NUM\_COLS \times 2/100.0 + NUM\_NODES/10000.0 \qquad (4\text{-}6)$$

$$wd\_fact = NUM\_ROWS \times 4/150.0 \qquad (4\text{-}7)$$

For designs with more than 2000 gates where the number of columns was less than the number of rows, we used Equation 4-8 to increase the height and Equation 4-9 to increase the width. In this case the height factor did increase, but the width factor is the same as Equation 4-5.

$$ht\_fact = NUM\_COLS \times 2/60.0 + NUM\_NODES/10000.0 \qquad (4\text{-}8)$$

$$wd\_fact = NUM\_ROWS \times 4/100.0 + NUM\_NODES/10000.0 \qquad (4\text{-}9)$$

These wire area equations were derived from an analysis of several different size layouts with different numbers of rows and columns. Using these equations for wire area along with the minimum height and width of a layout due to just the gates produced estimates within 18% of actual values for most designs.

To see how well the layout tool model and the area estimator performed on the test designs, we estimated height and width for each one. Then we compared these height and width estimates to those from the actual layouts. Table 9 shows the height and width estimates we obtained for the example test designs. This table shows the percent error between these estimates and the height and width values

from the actual layouts. All values except percent error are expressed in lambda units (1 lambda unit = 2 microns). This table shows that for these test designs, the height and width estimates are within 20% of the actual values. In addition, this table shows that the model works well for area estimation on designs other than adders.

Table 9 Area Estimates for Example Test Designs

Design Name	Number of Gates	Actual Height	Estimated Height	Percent Error	Actual Width	Estimated Width	Percent Error
add4	43	459	478	4.14%	303	298	-1.65%
add12	147	1123	1314	17.01%	891	979	9.88%
add24	303	1667	1593	-4.44%	1654	1950	17.90%
abs16	208	1283	1214	-5.38%	968	933	-3.62%
mult4	163	907	891	-1.76%	1472	1539	4.55%

4.6 Summary

This chapter described the details of implementing the area and delay estimation method. We used a small set of training designs based on adders and a small set of test designs (adders and non-adders) to show how the method works for modeling a target layout tool and using the model to estimate area and delay. Using these designs, we found delay estimates within 7% and area estimates within 18%. We found no discernable difference between estimating area and delay for adders or non-adders, even though the model was based on a training set with all adders. This is due to the fact that the non-adder designs are actually made up of adders so an adder-based training set produced good results. This chapter also described how we evaluated the performance of the learning system and the estimator. The next chapter describes the set of experiments we conducted to evaluate and validate the method using other training designs and larger test designs.

5.0 EXPERIMENTS AND RESULTS

Experimentation illustrates the identification of the general design features, the relative placement concepts, the formulation of the training set, the derivation of the tool model, and the application of this model to real world designs.

There are five major sets of experiments. The first set of experiments determines how well the learning system learns in this domain. This involves using abstract designs as training sets and "stacking the deck" with the concepts to be learned. This means manually assigning a relative placement concept to each training instance in an way such that a known relationship exists between the training instance and this concept. This way, the training sets are "stacked" with known relationships that map the features to the associated concepts. Now the task is to see how well the learning system finds these relationships. As was expected, it proved necessary to make adjustments to the learning system's biases so it could find the relationships and produce the appropriate rules. As described in Section 3.3.2.2, some of these adjustments include setting the range of certainty for acceptable rules and setting the maximum number of features that may appear in the left-hand side of each rule.

The second set of experiments validates the approach of using machine learning to model layout tools. This involves using training sets based on microprocessor ALU components, but the concepts are not "stacked". The relative placement concepts are derived from actual layouts. The experiments in this set address the following research issues:

- Is the machine learning method better than randomly assigning relative placement concepts?
- Is the machine learning method better than using one relative placement concept for all gate pairs?
- Is it possible to model the target layout tool using different configurations and/or execution parameters?

- Can the layout tool model handle a wide variety of design sizes?
- Do we have a sufficient number of test designs?
- Is using the model faster than producing actual layouts?

The third set of experiments shows how well the model works on real world benchmark designs. The generalized set of rules that model the layout tool's input-to-output relationships was applied to large and completely functional designs to obtain area and delay estimates. These estimates were then compared to area and delay values based on actual layouts of these designs and a measure of closeness was derived. This measure of closeness (percentage difference) shows how well the rules model the layout tool for non-trivial designs.

The fourth set of experiments shows the generality of the modeling method. Parts of the implementation were modified and another layout tool was modeled using the same method. The required modifications are described in detail and the resultant models of the two layout tools are compared and evaluated.

The fifth set of experiments shows where we stand in relation to other estimation techniques. We compare our approach using machine learning to other methods of estimating layouts and deriving area and delay estimates.

All of the experiments in this chapter (except for the benchmarks) were performed on a Sun SPARC2 workstation with 593 Mbytes. The benchmark experiments were performed on a Sun SPARC20 workstation with 1737 Mbytes. The learning system, Rule Learner, is installed on a DEC 3100 with 65 Mbytes.

5.1 Learning Results

This section discusses results from the first set of experiments. These results were obtained by using the modeling method on the ArtistII layout tool from the Keystone Tool Suite.

The purpose of the experiments described in this section is to determine if the learning system is capable of learning in the computer-aided design domain. For each of these experiments, a small test design was created and its layout was "stacked".

This means the layout was not obtained from a layout tool. Instead, the gates were placed manually according to their connectivity. This manual placement is based on the premise that if the gates are placed according to some order, the learning system should be able to learn the rules that represent this order.

One of the test designs created for this set of experiments consists of four separate circuits. Each circuit contains seven gates for a total of 28 gates (378 gate pairs). Each circuit has its own set of inputs and only one output. There are no connections between the four circuits. One circuit contains only 3-input NAND gates, another contains only 3-input NOR gates, another contains only 4-input NAND gates, and the last one contains only 4-input NOR gates. Figure 31 shows the graph for the 3-input NAND gate circuit. The graphs for the other three circuits are identical, except for the function of the gates and/or the number of inputs. The label inside each node in the figure is the designation of the gate as it is listed in the flattened description (.fla) file. These designations are also used in the grouping (.grp) file to indicate the placement of each gate.

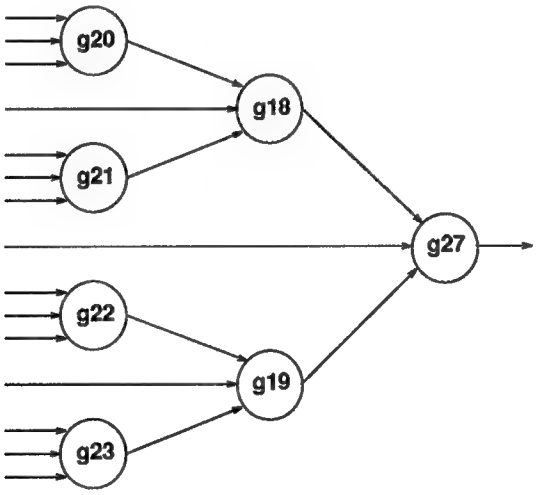


Figure 31 Test Design Graph

The "stacked" grouping file is shown in Figure 32. The designations in Figure 31 correspond to the first row in this file. To "stack" the layout, we manually placed each of the four circuits in its own row. In each row, we placed the circuit's external output gate in the middle. We placed the two gates that provide inputs to this gate

one on each side. We placed the remaining four gates on the ends of the row, two on each end.

```
(ts3stk
(row r1 g21 g20 g18 g27 g19 g22 g23 nil)
(row r2 g16 g17 g13 g26 g12 g14 g15 nil)
(row r3 g10 g11 g7 g25 g6 g8 g9 nil)
(row r4 g5 g4 g1 g24 g0 g2 g3 nil)
(column a1 r1 r2 r3 r4)
)
```

Figure 32 "Stacked" Placement File

Using this placement gives each of the four rows two pairs of gates (eight pairs total) that are near each other and share a common destination. These are the gates on the ends of the rows. In Figure 32, these are gate pairs g20-g21 and g22-g23. Gate pair g20-g21 share gate g18 as a common destination and gate pair g22-g23 share gate g19 as a common destination. We designed this arrangement to produce a rule that says eight of the gate pairs that are near share a common destination. There are also four pairs of gates on each row that are near each other and logically connected. In Figure 32, these are gate pairs g18-g20, g18-g27, g19-g22, and g19-g27. We designed this placement to produce rules that deal with the logically connected feature. This "stacked" placement also causes every gate on the same row to have the same function, area, and delay and causes gates on different rows to have different functions, areas, and delays. We expected the learning system to produce rules that reflect this.

Using this setup, we produced two training sets. One of the training sets used all features as described in Section 3.3.2.1 while the other used only relational features. We did this to see if individual gate features produced better rules that covered more gate pairs than relational features alone. Using only relational features, the learning system produced six rules. These are shown in Figure 33. (We did not calculate the certainty factors because we stacked the placement. In addition, because the training set was stacked, these rules have very low negative coverage.) These six rules covered a little more than half of the total number of training instances (192 out of 378).

As expected, the common destination feature showed up in one of the rules for the *near* concept. This rule indicated that eight out of the 24 total gate pairs with the concept of near shared a common destination. However, it also indicated that four out of the 354 gate pairs that are not near share a common destination. We examined the layout and found this was true. In each of the four rows, the gates on each side of the middle gate share a common destination, but are not near each other.

```
Found 6 Good Rules
192 examples covered (out of 378)

(log_conn in)  ==> near
 { pos=  0.33,  neg=  0.00,  cf=  0.000
  p=8, n=0, tp=24, tn=354 }

(com_dest yes)  ==> near
 { pos=  0.33,  neg=  0.01,  cf=  0.000
  p=8, n=4, tp=24, tn=354 }

(a1_a2 >)  ==> far
 { pos=  0.59,  neg=  0.00,  cf=  0.000
  p=172, n=0, tp=294, tn=84 }

(h1_h2 >)  ==> far
 { pos=  0.59,  neg=  0.00,  cf=  0.000
  p=172, n=0, tp=294, tn=84 }

(w1_w2 >)  ==> far
 { pos=  0.59,  neg=  0.00,  cf=  0.000
  p=172, n=0, tp=294, tn=84 }

(ni1_ni2 >)  ==> far
 { pos=  0.59,  neg=  0.00,  cf=  0.000
  p=172, n=0, tp=294, tn=84 }
```

Figure 33 Set of Rules Based on Stacked Placement

The logical connection feature also showed up in one of the rules for *near*. However, it was not as we expected. This rule indicated that eight of the gate pairs with the concept of *near* are logically connected with the first gate providing an input to the second gate ($log\_conn = in$). This is true in the "stacked" layout. However, there

are another eight gate pairs that are logically connected with the first gate receiving an output from the second gate (*log_conn* = *out*). This did not show up in the rules. It turns out that the learning system found both rules and opted to keep only one. We verified this by changing the admissible logical connection features values to *yes* and *no* instead of *in, out,* and *none.* All 16 gate pairs then showed up in a single rule. This happened because we effectively increased the generality of the logical connection feature. The target layout tool appears to treat connected gate pairs the same no matter which way they are connected. It does not matter if gate 1 provides an input to gate 2 or gate 2 provides an input to gate 1, this gate pair is considered connected and is treated as such.

No rules showed up for the concept of *same.* This was not expected at first, but further examination of the layout showed why. While it is true that all gate pairs with the concept of same have the same function and area, it is also true that the gate pairs where one gate is from the 3-input NAND circuit and the other gate is from the 3-input NOR circuit have the same area features. This is true for the gate pairs from the 4-input NAND and 4-input NOR circuits as well. These gate pairs have the concept of far and produce negative examples for the area features. Therefore, the negative threshold of a good rule was exceeded and the learning system did not produce the rule. In addition, there is currently no feature that indicates if the two gates in the pair have the same function, so rules indicating this cannot be produced.

Four rules showed up for the concept of *far.* Three dealt with relative area features and one dealt with the relative number of internal inputs. Each of the rules dealing with area indicated the proper attribute value (different areas, heights, and widths) and the correct coverage (172 out of 294). However, we did not expect to see the rule dealing with the relative number of internal inputs. We examined the 3-input NAND circuit graph (Figure 31) to see why it was produced. Three of the seven gates in this circuit have two internal inputs and one external input each (g18, g19, and g27) while the other four (g20, g21, g22, and g23) have only external inputs. This is true for each circuit and therefore each row. When the three gates with internal and external inputs from each row are paired up with the four gates with only external inputs from the other three rows, the number of internal inputs will be different. This produced the rule dealing with the relative number of internal inputs.

Running the learning system after including individual gate features in the training set produced 19 more rules and more coverage. As we expected, the same relational feature rules as before showed up. However, the majority of additional rules dealing with individual gate features made no sense, even with the "stacked" layout. Each of these rules had low coverage and therefore did not reflect any real patterns in the layout. This suggests that when dealing with gate pairs, only features that describe how the two gates relate to each other should be used.

We have good confidence that the results from these "stacked" experiments and others indicate the learning system is very capable of learning in this domain.

5.2 Validation Results

For the validation experiments, we used a set of simple microprocessor components as training and test designs. These designs are listed in Table 10, showing each design's name, function, the number of nodes (primitive gates) required, and if the design was used as a training design, test design, or both. Because the training sets were always made up of several training designs, we did not run the risk of overfitting the learning process by training and testing with the same design. We only used one design at a time to test the model's performance. We believe these designs represent a varied cross-section of different types and sizes of microprocessor components. (Later in this section, we perform a statistical analysis to show that these designs represent a sufficient sample.)

We obtained these designs by using the TinkerTool [65] system. The TinkerTool system automatically generates VHDL descriptions of the designs, processes them, and eventually produces layouts using the ArtistII layout tool. TinkerTool also runs Magic to obtain the height and width of the layout and IRSIM [66] to obtain the critical path delay. We used these as the actual area and delay for error calculations. Each test design has its own actual height, width, and delay.

Before we describe the experiments and their results, we need to identify how we measure the performance of the estimation method using machine learning to model layout tools. There are two main sources of error in this approach. One source of error comes from applying the layout tool model to target designs. This error

Table 10 Set of Training/Test Designs

Design Name	Function	Number of Gates	Number of Transistors	Type
add4	4-b adder	43	146	Training/Test
add8	8-b adder	95	330	Training/Test
add12	12-b adder	147	514	Training/Test
add16	16-b adder	199	698	Training/Test
add24	24-b adder	303	1066	Test
add32	32-b adder	407	1434	Training/Test
abs16	16-b absolute value	208	638	Test
inc4	4-b incrementer	23	68	Test
inc8	8-b incrementer	51	156	Test
inc16	16-b incrementer	107	332	Test
sub4	4-b subtracter	48	160	Test
sub8	8-b subtracter	104	352	Test
sub16	16-b subtracter	216	736	Test
addsub4	4-b adder/sub	64	224	Test
addsub8	8-b adder/sub	132	464	Test
addsub16	16-b adder/sub	268	944	Test
mult2	2-b multiplier	40	112	Test
mult4	4-b multiplier	163	550	Test
mult6	6-b multiplier	389	1378	Test
mult8	8-b multiplier	711	2574	Test
mult16	16-b multiplier	2959	11038	Test

is the aggregation of certainty factors, the order of relative placement concepts in the gate pair list, and how well the learning system learned from the training set. A figure of merit to evaluate this error compares area and delay estimates using the estimator on layout information from an actual layout and using the estimator on layout information from applying the model. This figure of merit is called the *modeling error.*

The other source of error is due to the estimator itself. This estimator uses models of wire area and wire delay to derive area and delay estimates from layout information. The wire area model uses statistical data to provide the average wire length between positions in the layout and the required area for routing the wires among the gates. The wire delay model provides a worse case delay due to the wires between gates. A figure of merit to evaluate this error compares the actual area and

delay (from using Magic and IRSIM on an actual layout) to area and delay estimates from using the estimator on information from an actual layout. This is called the *estimator error.*

Another figure of merit compares area and delay estimates from using the estimator on layout information from applying the model to actual area and delay from using Magic and IRSIM on an actual layout. This is called the *overall error.* This error is a combination of the *modeling error* and the *estimator error* because we use the estimator on layout information produced by applying the model to target designs.

The *overall error* shows how well the method works overall and the *modeling error* shows how well the model produced by the learning system performs. The *estimator error* shows how well the area and delay estimator performs on layout information. Now that we have defined the figures of merit for measuring performance, we can describe the experiments.

The first experiment in validating the method determines if machine learning is better than randomly assigning relative placement concepts. For this experiment, we used three test designs: an 8-bit adder, a 16-bit adder, and a 4-bit multiplier. The adders provided two of the same type of design with different sizes and the multiplier provided a different design type, also with a different size. Table 11 shows the overall error in delay estimates for machine learning and randomly assigning concepts. For the 8-bit adder, machine learning produced better delay estimates. For the 16-bit adder, both machine learning and randomly assigning concepts had about the same overall error. This indicates that in some cases randomly assigning concepts may come close to classifying the correct concepts. For the 4-bit multiplier, we see that randomly assigning concepts produces a much worse error than machine learning does.

Table 11 Machine Learning vs. Random Concept Assignments for Delay Estimates

Design Name	Number of Gates	Machine Learning	Random Assignments
add8	95	-3.68%	5.07%
add16	199	3.43%	3.42%
mult4	163	-1.41%	27.20%

We also looked at the modeling error in area estimates for these three designs. Table 12 shows a comparison of the modeling errors for height and width from machine learning and randomly assigning concepts. In a few of the cases but not all, randomly assigning concepts produced a better estimate. This is to be expected because sometimes a guess may be closer than a well thought out calculation using estimates. However, this table shows that randomly assigning relative placement concepts does not consistently produce better estimates than machine learning.

Table 12 Machine Learning vs. Random Concept Assignments for Area Estimates

Design Name	Number of Gates	Machine Learning (height)	Machine Learning (width)	Random Assignments (height)	Random Assignments (width)
add8	95	5.11%	-2.56%	7.96%	-9.50%
add16	199	-5.40%	0.29%	-3.69%	9.78%
mult4	163	8.55%	-0.86%	3.09%	2.91%

The next experiment determines if using one relative placement concept for all gate pairs is better than machine learning. Using one concept means that we manually assign the same concept to each gate pair. For this experiment, we used the concepts of *near_s, close_1*, and *far_1*. The concept of *near_s* indicates the two gates are located next to each other on the same row. The concept of *close_1* indicates the two gates are located two to three positions apart. The concept of *far_1* indicates the two gates are located six to seven positions apart. Table 13 shows a comparison of the overall error for delay estimates from machine learning and from using one concept for all gate pairs. This table shows that in the case of delay, machine learning produces better estimates. For example, the 4-bit multiplier has an overall error of -1.41% for delay when machine learning is used. It has an overall error of -30.78% when all connected gate pairs are assigned the concept of *near_s*, an overall error of -23.72% when all gate pairs are assigned the concept of *close_1*, and an overall error of -9.60% when all gate pairs are assigned the concept of *far_1*. Each of these errors is much worse than the overall error from assigning the concepts through machine learning.

Table 14 shows the results from this experiment using height estimates. The numbers indicate the modeling errors for height estimates. In the case of the 8-bit adder, machine learning produced better height estimates. However, for the other

Table 13 Machine Learning vs. One Concept for Delay Estimates

Design Name	Number of Gates	Machine Learning	One Concept (near_s)	One Concept (close_1)	One Concept (far_1)
add8	95	-3.68%	-38.61%	-31.72%	-17.93%
add16	199	3.43%	-52.80%	-47.62%	-37.28%
mult4	163	-1.41%	-30.78%	-23.72%	-9.60%

two designs, machine learning did not produce better estimates. At first glance, this may indicate that using one concept is better than machine learning. However, this table only shows the modeling errors for height estimates. When we consider the modeling errors for width estimates, the story changes. Table 15 shows the results for width estimates. In most of the cases here, machine learning produced much better estimates. Because height and width are two inseparable components of area, we must consider them together. Table 16 shows the modeling errors for total area estimates based on machine learning and on using only one concept. This table shows that in most cases, machine learning provides better total area estimates than assigning one concept to all gate pairs.

Table 14 Machine Learning vs. One Concept for Height Estimates

Design Name	Number of Gates	Machine Learning	One Concept (near_s)	One Concept (close_1)	One Concept (far_1)
add8	95	5.11%	7.96%	7.96%	10.81%
add16	199	-5.40%	-2.46%	-1.23%	0.00%
mult4	163	8.55%	5.82%	3.09%	3.09%

Table 15 Machine Learning vs. One Concept for Width Estimates

Design Name	Number of Gates	Machine Learning	One Concept (near_s)	One Concept (close_1)	One Concept (far_1)
add8	95	-2.56%	-0.49%	-2.68%	-11.69%
add16	199	0.29%	2.25%	-4.21%	-5.38%
mult4	163	-0.86%	-5.36%	-1.26%	-2.71%

Another issue brought out in this research is if the modeling method can be used to model the target layout tool using different configurations and/or execution parameters. For example, we can change the number of iterations for the ArtistII

Table 16 Machine Learning vs. One Concept for Total Area Estimates

Design Name	Number of Gates	Machine Learning	One Concept (near_s)	One Concept (close_1)	One Concept (far_1)
add8	95	2.42%	7.43%	5.07%	-2.15%
add16	199	-5.12%	-0.27%	-5.39%	-5.38%
mult4	163	2.62%	-4.50%	-2.93%	-4.36%

layout tool. A higher number of iterations produces layouts with less area because the cost function is based on area. This issue was raised when we tried to run ArtistII on the 16-bit multiplier design using 10,000 iterations. ArtistII ran for over two weeks with no results before the host machine had to be rebooted. The required length of time to produce a layout for this design using 10,000 iterations was too long, so we had to reduce the number of iterations. We reduced the number to 100 and got a layout in 28,746 CPU seconds, which is close to eight hours. However, we realized that using a model based on 10,000 iterations to produce area and delay estimates and then comparing these estimates against actual values from a real layout based on only 100 iterations would be an invalid comparison. Therefore, we needed to build another training set based on training design layouts using only 100 iterations of ArtistII.

We used the same five training designs as before (4-bit adder, 8-bit adder, 12-bit adder, 16-bit adder, and 32-bit adder) but produced layouts for each one using ArtistII running only 100 iterations. The learning system processed this training set and produced a set of rules in 144.4 CPU seconds. The results of this experiment are shown in Table 17. The actual values in the second column come from an actual ArtistII layout using only 100 iterations. The third column shows the estimated values from applying our 10,000 iteration based model in place of ArtistII. The fourth column shows the overall errors between the 10,000 iteration model and using ArtistII at 100 iterations. The fifth column shows the estimated values from applying our 100 iteration model. The sixth and last column shows the overall errors between the 100 iteration model and using ArtistII at 100 iteratons.

Table 17 shows that the estimates from the model using 100 iterations are better than the estimates from the model using 10,000 iterations. The width estimates are

Table 17 Comparison of Using Different Models on 16-Bit Multiplier

Physical Dimension	Actual Value	Estimated Value (10,000)	Overall Error	Estimated Value (100)	Overall Error
delay	994.8	1065.32	7.09%	1041.48	4.69%
height	18755	14864	-20.75%	15643	-16.59%
width	9921	9970	0.49%	9970	0.49%
area	186068355	148194080	-20.35%	155960710	-16.18%

the same because the widest row in the estimated layout for both models happened to have the same width. The height estimate using the 10,000 iteration model is underestimated because the model is based on 10,000 iterations of ArtistII which should produce a smaller layout. This also causes the total area to be underestimated. This experiment shows that it is possible to create different models for different configurations and/or execution parameters for the target layout tool. This experiment also shows that the different models do reflect different configurations of the target layout tool.

The next set of results addresses the issue of using the layout tool model on a wide variety of design sizes. We wanted to determine how well the layout tool model worked and how well the overall estimation method worked. Eight graphs are used to show the results from applying the layout tool model to several test designs. These graphs show percentage error vs. the wide range of design sizes found in the set of test designs. (In each of these graphs, the horizontal axis is nonlinear due to the wide range of varying design sizes.) To obtain these results, we used five training designs and built two training sets. We used a 4-bit adder, an 8-bit adder, a 12-bit adder, a 16-bit adder, and a 32-bit adder. One training set was built using relative placement concepts from ArtistII layouts using 10,000 iterations. (This is the same training set as described in Section 4.1). The other training set was built using relative placement concepts from ArtistII layouts using only 100 iterations. We needed the 100 iteration model because it took too long using 10,000 iterations to produce actual layouts for the large designs. We had to run ArtistII with only 100 iterations to get the actual layouts to compare against. Therefore, in order to make a valid comparison, we had to produce a model of ArtistII using only 100 iterations.

We processed each of these training sets through the learning system and produced a set of rules for each one. These two sets of rules became our two models. (It took 149.8 CPU seconds to produce the 10,000 iteration model and 144.4 CPU seconds to produce the 100 iteration model. A more in-depth comparison of these two models is provided later.) We applied the 10,000 iteration model to each of the smaller test designs (those to the left of the vertical line on the graphs) to obtain predicted layout information. We used our estimator and this predicted layout information to obtain height, width, area, and delay estimates for each test design. Likewise, we applied the 100 iteration model to the larger designs (those to the right of the line) in the same manner. It should be noted that ArtistII requires a larger number of iterations for larger designs in order to produce better layouts. Using a reduced number of iterations may produce layouts in a reasonable period of time (CPU hours vs. CPU weeks), but the layouts are not very good in terms of area. Therefore, even though it provides a valid comparison with ArtistII using 100 iterations, the 100 iteration model will not produce good results.

Because each test design has its own actual area and delay, we are able to determine the modeling error and the overall error. The little "x" symbols in each graph indicate the percentage error vs. the number of nodes in the design. For eample, Figure 34 shows a graph of the modeling error for height estimates. (It should be pointed out again that the modeling error indicates how well the model performs. It does not include any errors due to the estimator.) This graph shows that the modeling error is well within 10% for most of the design sizes. This graph also shows that some of the smaller designs are outside the 10% limit. This is because small differences in small numbers create bigger percentage errors.

Figure 35 shows a graph of the modeling error for width estimates. This graph shows that the modeling error here is also well within 10% for all of the designs. Figure 36 shows the modeling error for total area estimates. This graph shows that most of the designs have a modeling error within 10%. The other designs' modeling errors are outside this range because the individual modeling errors for height and width did not offset each other and added up in one direction. However, these errors are still well within 20%. Figure 37 shows a graph of the modeling error for delay estimates. This graph shows that the modeling error for delay is within 10% for

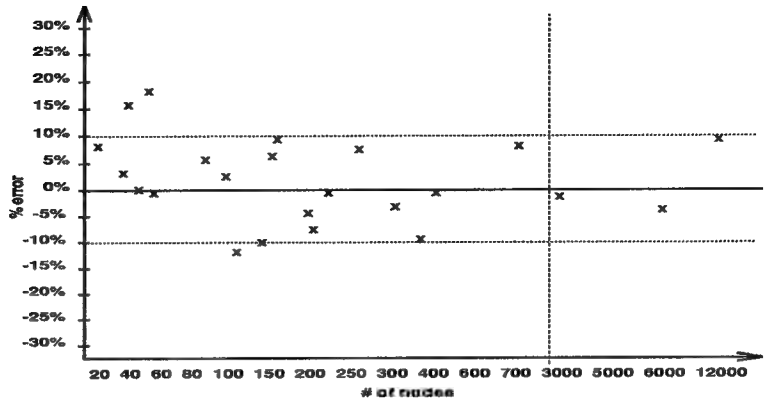


Figure 34 Modeling Error for Height Estimates

designs up to 3000 gates, but then it starts to drop off to around -15%. This is due to larger designs having more gates on the critical path and the errors due to the wire delay model start to add up. However, these errors are still within 30%.

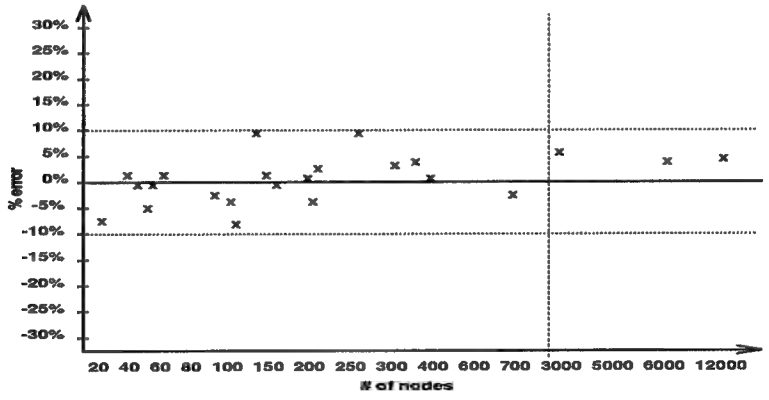


Figure 35 Modeling Error for Width Estimates

These last four graphs show that the model performs quite well for a wide range of design sizes. The next four graphs look at how the overall method performs for these designs. (It should be pointed out again that the overall error includes the modeling error and the estimator error.)

Figure 38 shows a graph of the overall error for height estimates. This graph shows the overall error for most designs falls within 10%. However, in a few of the small to medium designs, the height is overestimated. This indicates these designs

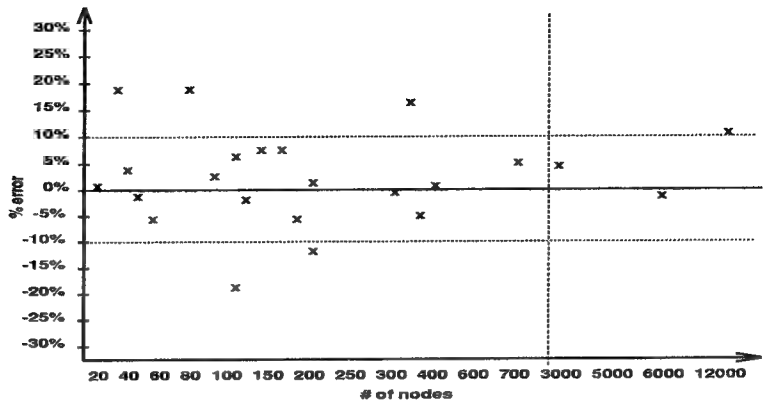


Figure 36 Modeling Error for Area Estimates

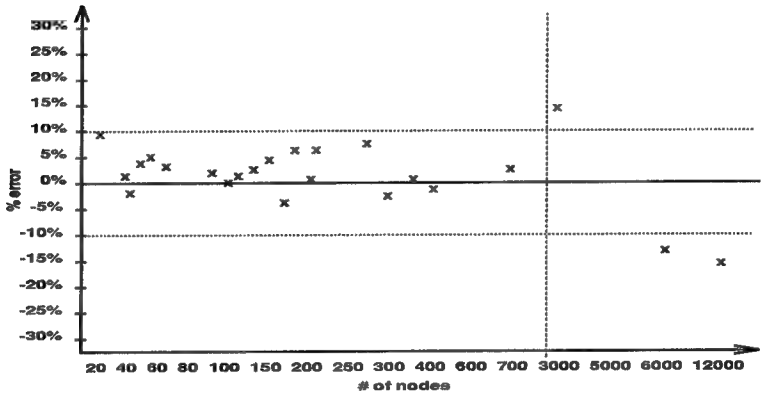


Figure 37 Modeling Error for Delay Estimates

have more horizontal wires routed over the gates so the wire area model overestimates the horizontal wire area. This graph also shows that the height is underestimated for large designs. In these designs, there are more horizontal wires between gates that require more wire area.

Figure 39 shows a graph of the overall error for width estimates. This graph also shows that most of the designs are within 10%, with only a few designs outside this range. This indicates these designs have vertical wires routed over gates so the wire area model overestimates the vertical wire area. However, all of the width estimates are still within 30%. Figure 40 shows a graph of the overall error for total area estimates. Here again, most of the overall error is within the 10% range. However,

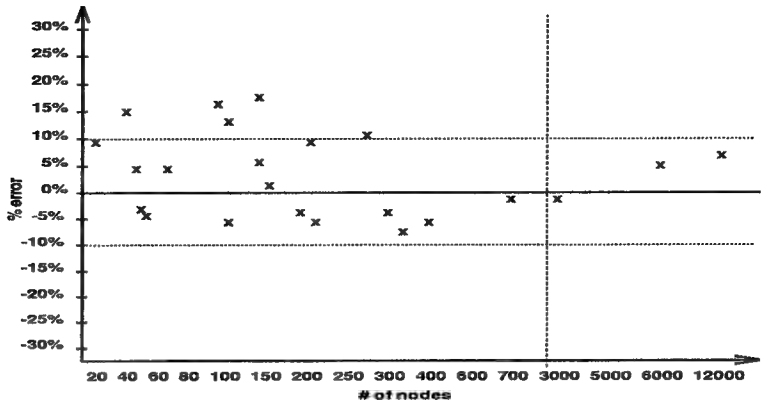


Figure 38 Overall Error for Height Estimates

a lot of designs have their overall errors outside this range. This is because the individual height and width errors did not offset each other and added up in one direction. But, even with this happening, all of the errors are still within our goal of 30%.

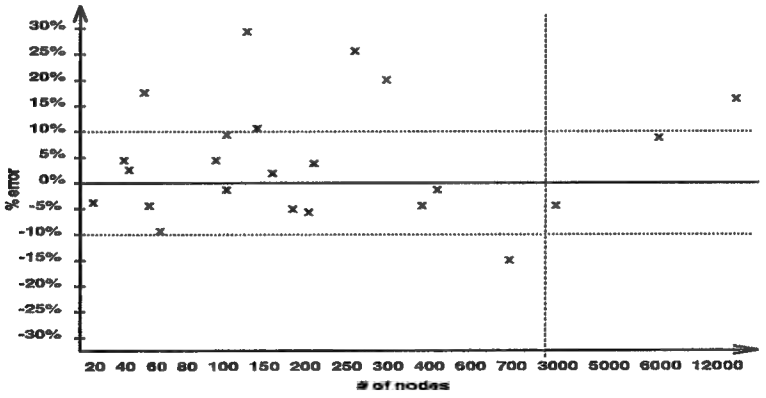


Figure 39 Overall Error for Width Estimates

Figure 41 shows a graph of the overall error for delay estimates. This graph shows the delay estimates for most designs are also within 10%. However, there are a few designs outside this range. The design with close to 6000 gates has an underestimated delay around -30%. If we look back at Figure 37 we can see that this design had a underestimated modeling error of about -15%. This modeling error combined with the error from the wire delay model underestimated the delay even more.

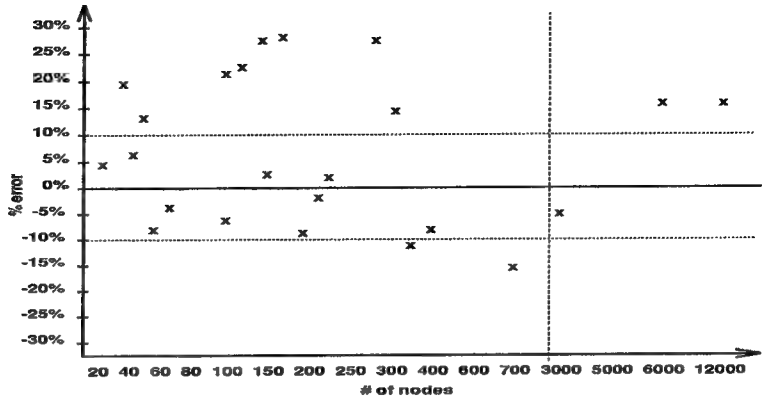


Figure 40 Overall Error for Area Estimates

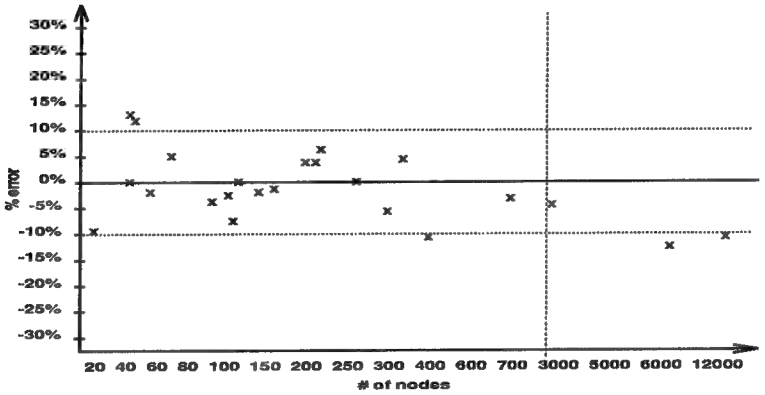


Figure 41 Overall Error for Delay Estimates

At this point, a comparison of the 10,000 iteration model and the 100 iteration model is warranted. Table 18 compares the 10,000 iteration model of ArtistII to the 100 iteration model. Although there are the same number of instances in each model (1144), there is different supporting evidence for each of the concepts. The number of instances are the same because both models are based on the same set of training designs. The supporting evidence is different because the actual layouts for each training design are different.

For example, in the 10,000 iteration model, we have 123 instances with the concept of *close_1* and 9 rules. In the 100 iteration model, we have 231 instances with the concept of *close_1* and 18 rules. For this concept, there is twice as much support in

Table 18 Comparison of Two ArtistII Models

Relative Placement Concept	Number of Instances (10,000 Its.)	Number of Rules (10,000 Its.)	Number of Instances (100 Its.)	Number of Rules (100 Its.)
near_s	82	7	101	8
near_d	20	0	4	1
close_1	123	9	231	18
close_2	130	5	74	6
far_1	119	6	68	6
far_2	85	0	82	0
way_out	585	8	584	8
total	1144	35	1144	47

the 100 iteration model as there is in the 10,000 iteration model. Table 18 shows similar comparisons for the remaining concepts.

As far as the contents of the rules, there are many differences. For example, there is a rule in the 10,000 iteration model that indicates a concept of *far_1* for any gate pair that have an unequal number of internal inputs and the same function. This rule has a certainty factor of 0.639. There is a rule in the 100 iteration model that indicates a concept of *close_1* for the gate pairs meeting the same conditions with a certainty factor of 0.900. This is because there is more evidence in the 100 iteration model's training set for this relationship. Our examination of the 10,000 iteration model and the 100 iteration model shows they are indeed two different models.

Now that we have determined how well the model performs and how well the overall estimation method performs, we can answer a big question raised in proposing this research. This question is: "What percentage error in area and delay estimation is both achievable and acceptable?" As described in the introduction to this thesis, a paper on layout estimation that Nourani and Papachristou presented at the 30th ACM/IEEE Design Automation Conference [26] in 1993 states a difference of less than 10% is considered very good, but difficult to achieve. In addition, the VLSI CAD community considers a difference of 20-30% to be both acceptable and achievable [67]. Therefore, this research adopted the goal of trying to achieve less than 30% difference while attempting to approach 10% as much as possible. It is evident from these graphs that we achieved our goal of less than 30% difference with the majority

of estimates falling within 10%. This is true for both the layout tool model and the overall estimation method.

Another key issue is how well our set of test designs samples the population space of designs. To address this issue, we calculated a confidence interval [68, 69]. A confidence interval shows how well our set of test designs samples the population space and indicates if it is a truly representative sample. For this analysis, we used the number of gates for the first 20 designs in Table 10. (We did not include the 16-bit multiplier because its large size perturbated the results.) The confidence interval was found using Equation 5.2, where $\bar{x}$ is the mean, σ is the standard deviation, t is the critical value, and n is the number of samples. The critical value came from a table [69] and is based on the degrees of freedom and the desired confidence level. For a sample size of 20, there are 19 degrees of freedom. If we wish to use a confidence level of 99%, the critical value t is 2.86.

$$Confidence\ Interval = \bar{x} \pm \frac{\sigma t}{\sqrt{n}} \tag{5-1}$$

For our set of test designs, $\bar{x} = 185.9$, $\sigma = 163.4$, and $n = 20$. Using these values in Equation 5.2 gives us a confidence interval of 81.4 to 290.4, at a 99% confidence level. This indicates that we have sufficiently sampled a population of designs with a mean number of gates between 81.4 and 290.4. In order to sample a population with a higher mean, we would have to use more test designs with more gates. However, we are limited to test designs using 32 bits or less due to restrictions in our VHDL compiler. This means we would have to build several test designs using combinations of smaller designs.

Another worthwhile analysis looks at the densities of the ArtistII layouts versus the densities of our estimated layouts. Layout density is defined as the number of gates per unit area. To find the density of a given layout, we divide the number of gates by the total area. For example, if a design has 512 gates and a total area of 15000 sq. microns, the density would be 0.0341 (512/15000). For this analysis, we examined both our suite of test designs and the benchmark designs. (The benchmark designs are described in Section 5.3.) For our test designs, the average density of the

ArtistII layouts is 187.70E-6, while the average density for our estimated layouts is 176.44E-6. The average difference between the densities of ArtistII layouts and our estimated layouts is -3.25%. This shows that our estimated layouts have densities very close to that of ArtistII layouts.

For the benchmark designs, the average ArtistII layout density is 30.55E-6. (This indicates the benchmark designs are less dense than the test designs, i.e., there are fewer gates per unit area. This makes sense because the benchmark designs have a lot more gates and the total area is mostly due to interconnection wiring.) The average density for our estimated layouts is 33.70E-6. The average difference between the densities of the ArtistII layouts and our estimated layouts for the benchmarks is 7.28%. This shows that even for large complex designs, our estimated layouts have very similar densities.

The final set of results in this section shows that using the layout tool model is much faster than producing an actual layout. In fact, these results show that pre-processing the target design, applying the model, and even deriving area and delay estimates can be performed much faster than just producing an actual layout. Table 19 shows these execution times for a wide variety of test designs. The first column shows the test design. The second column shows the size of the design by the number of gates. The next three columns show the time it takes to pre-process the design, apply the model to produce a list of gate pairs and their predicted relative placement concepts, and to derive area and delay estimates. The area and delay estimate time includes producing an estimated layout and exhaustively searching each design twice to find the critical path for both high-low and low-high transitions on the critical path's output. The next to last column shows the total time for using the model which is the sum of the times for pre-processing, applying the model, and deriving estimates. The last column shows the time it takes for the ArtistII layout tool to just produce a layout. Layouts for all of the test designs except for the last three were produced by ArtistII using 10,000 iterations. Due to their large size, the last three designs in the table, *mult16*, *big*, and *realbig* required a reduced number of iterations (100) in order for ArtistII to produce a layout at all! In order to keep the comparisons valid, we built a different training set that models ArtistII using only 100 iterations and used the resultant model to obtain these execution times.

Table 19 Execution Time Comparison (CPU sec)

Design Name	Number of Gates	Pre-Processing	Applying Model	Estimating Area and Delay	Total Time	ArtistII Time
add4	43	0.13	0.2	0.16	0.49	840
add8	95	0.29	0.3	0.37	0.96	2278
add12	147	0.52	0.4	0.78	1.70	5385
add16	199	0.96	0.7	1.16	2.82	8278
add24	303	1.91	1.5	2.41	5.82	11886
add32	407	3.12	3.6	3.99	10.71	18735
abs16	208	0.85	0.6	1.26	2.71	6676
inc4	23	0.40	0.1	0.11	0.61	405
inc8	51	0.80	0.2	0.15	1.15	980
inc16	107	0.27	0.3	0.40	0.97	3408
sub4	48	0.10	0.2	0.18	0.48	828
sub8	104	0.30	0.4	0.48	1.18	2374
sub16	216	1.70	0.6	1.37	3.67	7788
addsub4	64	0.15	0.4	0.25	0.80	1398
addsub8	132	0.42	0.4	0.62	1.44	4896
addsub16	268	1.45	1.0	1.79	4.24	10224
mult2	40	0.90	0.2	0.13	1.23	653
mult4	163	0.63	0.7	0.89	2.22	4106
mult6	389	3.14	3.5	3.68	10.32	25213
mult8	711	10.39	7.2	11.44	29.03	47696
mult16	2959	190.47	56.9	195.20	442.57	28746
big	6264	938.84	217.6	1059.00	2215.44	65493
realbig	12528	3852.49	879.4	2873.84	7605.73	359289

It is evident from this table that using the layout tool model is orders of magnitude faster than just producing an actual layout. For example, the *mult8* test design, an 8-bit multiplier, required 47,696 CPU seconds for ArtistII to produce a layout. Using the layout tool model instead only requires 29.03 CPU seconds, and this includes deriving area and delay estimates. Even for large designs like *realbig*, using the layout model is much faster than running ArtistII. This is especially important when you consider that we only used 100 iterations of ArtistII for the large designs. Overall, combining these execution time results with the area and delay estimate results shows that this method of modeling layout tools does produce accurate estimates very quickly.

5.3 Benchmark Results

This section describes the results from applying our model to benchmark designs. (These benchmarks were obtained through MCNC, Center for Microelectronic Systems Technologies, in North Carolina [70].) Because these benchmark designs each contain more than 2000 gates, we had to use a reduced number of iterations (100) of ArtistII just to get a layout. Therefore, we used our model of ArtistII at 100 iterations to derive the area and delay estimates. Table 20 shows the actual height and width from the ArtistII layouts and compares these to the height and width estimates from using our model. These results show that we are within our goals for height and width estimates on the benchmark designs.

Table 20 Height and Width Estimates for Benchmark Designs

Design Name	Number of Gates	Actual Height	Estimated Height	Percent Error	Actual Width	Estimated Width	Percent Error
GCD16	2004	8666	9833	13.47%	7233	6084	-15.89%
DIV16	2085	8858	7376	-16.73%	6316	6317	0.02%
HAL8	3621	9235	8052	-12.81%	9627	8191	-14.92%
GCD32	3669	10747	11339	5.51%	13246	11532	-12.94%
DIV32	4177	14299	16361	14.42%	10383	11384	9.64%
HAL16	8416	26690	24680	-7.53%	16214	18368	13.28%

Table 21 shows the actual delay obtained through IRSIM simulations of the benchmark designs and the delay derived from information produced by applying our model of ArtistII to the benchmark designs. It should be noted here that up until now, our method has only been tested on purely combinational circuits. These results show that our method is applicable to and performs very well on sequential circuits as well as combinational circuits.

Table 22 shows a comparison of the execution times for our method and for using ArtistII to obtain a layout. The times for using our method include pre-processing the design, applying the model, and estimating area and delay. The ArtistII execution times are based on 100 iterations. These results show that our method is orders of magnitude faster than just producing a layout using ArtistII with a reduced number of iterations, even for large complex designs.

Table 21 Delay Estimates for Benchmark Designs

Design Name	Number of Gates	Actual Delay (ns)	Estimated Delay (ns)	Percent Error
GCD16	2004	158.9	129.51	-18.49%
DIV16	2085	208.6	199.71	-4.26%
HAL8	3621	195.9	176.68	-9.81%
GCD32	3669	331.7	327.78	-1.18%
DIV32	4177	344.7	336.15	-2.48%
HAL16	8416	749.6	686.31	-8.44%

Table 22 Execution Time Comparison for Benchmark Designs (CPU sec)

Design Name	Number of Gates	Pre-Processing	Applying Model	Estimating Area and Delay	Total Time	ArtistII Time
GCD16	2004	36.88	18.27	75.63	130.78	6513
DIV16	2085	38.16	11.52	92.22	141.90	5948
HAL8	3621	126.41	51.46	283.49	461.36	13940
GCD32	3669	415.03	60.09	282.45	757.57	54213
DIV32	4177	507.50	68.85	375.45	951.80	50293
HAL16	8416	2015.30	305.58	1602.84	3923.72	217432

5.4 Modeling Other Layout Tools

This section describes how the modeling method can be used to model a different layout tool. This time the target layout tool is TimberWolf. The TimberWolf layout tool is described in Section 2.2.3. This section also presents results from using the TimberWolf model to estimate area and delay. These results are presented at the end of this section.

Because this is a different layout tool, we had to implement the method in a different way. The method itself was not changed, just the implementation details. TimberWolf uses standard cells rather than primitive gates, so we modified the implementation to handle standard cells. The node library that provides information on the primitive gates was replaced with a node library that provides the same information for standard cells. As described in Section 3.3.2.1, the modeling method requires a physical characteristics library that provides information such as a cell's total area, height, width, intrinsic delay (for both high-low and low-high transitions),

the number of inputs, and the cell's function.

Another modification was changing the parsing routine in the Match tool. Instead of parsing a flattened file description, the parsing routine has to parse a cell-based netlist. This netlist provides the same type of interconnection and topology information that the flattened file description provides. The parser also had to be modified to parse the layout description file from TimberWolf. This file provides the same type of relative placement information that the .grp file does for ArtistII.

The wire delay model and wire area model also require modification. A standard cell layout is different from a layout using primitive gates, so these models must reflect this. The wire delay model is still a function of the length of wire connected to the output and the number of loads on the output, but now it must be based on standard cells. Because we are using standard cells, the wire area model can be based on the required channel width. This means a wire area model such as Rent's rule [21] or Left Edge Algorithm [59] can be used with good results.

Once these modifications were made, the modeling process is the same. We build training sets using the same training designs, but with training instances based on cell pairs (instead of gate pairs) and relative placement information from layouts produced by TimberWolf. The learning process is the same as well. The learning system trains on the training set and produces a set of rules. The set of rules becomes the model of the input-to-output relationships that result from using TimberWolf. Applying the model to obtain layout information is done in the same way as before. The target designs are pre-processed and the rules are applied. A list of cell pairs and their predicted relative placement concepts is generated. This list provides the necessary layout information so area and delay can be estimated.

After making modifications to the parsing routine and creating a new node library based on standard cells, we produced a model of TimberWolf. It only took a couple of hours to make the modifications to the parsing routine and producing the model of TimberWolf only took 1008 CPU seconds. The training set is based on five adder designs: a 4-bit, 8-bit, 12-bit, 16-bit, and 32-bit adder. The training instances are based on cell pairs that are logically connected. Relative placement concepts came

from layouts produced by TimberWolf. We also used the same reduced Partial Domain Model as we did in Section 4.1. This Partial Domain Model is copied here in Figure 42 for easy reference.

```
near_s, near_d, close_1, close_2, far_1, far_2, way_out. | concepts
com_dest:       yes, no.
com_src:        yes, no.
a1_a2:          ==, !=.
h1_h2:          ==, !=.
w1_w2:          ==, !=.
nt1_nt2:        ==, !=.
nx1_nx2:        ==, !=.
ni1_ni2:        ==, !=.
fun1_fun2:      ==, !=.
name:           ignore.
```

Figure 42 Partial Domain Model for TimberWolf

Using this training set, the learning system produced 29 rules. Figure 43 shows six of the 29 rules, with one rule for each concept. No rules were produced for the concept of *far_2*. This is because there was no supporting evidence in the training set for this concept. We compared these rules to those we obtained for the ArtistII model and found some differences.

Table 23 compares the models for ArtistII and TimberWolf. First of all, the distribution of concepts is different for the two models. The logically connected cells in the TimberWolf layouts tend to be placed closer together than the gates in the ArtistII layouts. The TimberWolf training set has 237 instances of cell pairs being placed next to each other on the same row (concept of *near_s*). This is close to half of the total number of training instances (620). This occurs because the TimberWolf layout tool fine tunes the placement in its third and last stage of execution so less wire area is required. (The three stages of the TimberWolf layout tool are explained in Section 2.2.3.)

The ArtistII training set has only 82 instances of gate pairs having the concept of *near_s*. This is only about 7% of the total number of training instances (1144). On the other end of the relative placement spectrum, the ArtistII training set has 585

```
(com_dest no) (fun1_fun2 !=)  ==> near_s
 { pos=  0.78,  neg=  0.49,  cf=  0.616
  p=186, n=186, tp=237, tn=383 }

(nx1_nx2 ==) (ni1_ni2 ==)  ==> near_d
 { pos=  0.27,  neg=  0.08,  cf=  0.702
  p=6, n=50, tp=22, tn=598 }

(nx1_nx2 !=) (ni1_ni2 ==)  ==> close_1
 { pos=  0.26,  neg=  0.13,  cf=  0.646
  p=35, n=65, tp=137, tn=483 }

(fun1_fun2 ==)  ==> close_2
 { pos=  0.18,  neg=  0.06,  cf=  0.734
  p=18, n=30, tp=101, tn=519 }

(com_dest yes)  ==> far_1
 { pos=  0.50,  neg=  0.30,  cf=  0.611
  p=29, n=171, tp=58, tn=562 }

(com_dest yes) (nt1_nt2 !=)  ==> way_out
 { pos=  0.49,  neg=  0.30,  cf=  0.604
  p=22, n=174, tp=45, tn=575 }
```

Figure 43 Example Set of Rules for TimberWolf

instances of gate pairs being located 10 or more positions apart (concept of *way_out*). This is a little over half of the total number of training instances. The TimberWolf training set has only 45 instances for the concept of *way_out*. This is about 7% of the total number.

The distribution of rules among the concepts for both models is about the same. This shows there is about an equal amount of supporting evidence for each of the concepts in both models. (Except for the concepts of *near_d* and *far_2* in the ArtistII model and the concept of *far_2* in the TimberWolf model which shows there is no supporting evidence for these concepts.) A roughly even amount of supporting evidence indicates the set of training designs adequately represents a good sample of each concept. This also indicates that the concepts chosen and the scope of the concepts sufficiently handles the necessary relative placement information.

Table 23 Comparison of ArtistII and TimberWolf Models

Relative Placement Concept	Number of Instances (ArtistII)	Number of Rules (ArtistII)	Number of Instances (TimberWolf)	Number of Rules (TimberWolf)
near_s	82	7	237	5
near_d	20	0	22	5
close_1	123	9	137	5
close_2	130	5	101	5
far_1	119	6	58	1
far_2	85	0	20	0
way_out	585	8	45	8
total	1144	35	620	29

While some of the rules are the same for both models, the majority of the rules are different. Having some of the rules the same in both models is expected because we are using the same set of training designs and the same Partial Domain Model (design features) for both models. It is also expected that more rules are different because these two target layout tools operate at different levels and their layout styles are different. For example, there is a rule in both models that indicates a concept of *near_s* for any two logically connected gates or cells that do not share a common destination and have different functions. The certainty factor for this rule in the ArtistII model is 0.603 and the certainty factor in the Timberwolf model is 0.616. These certainty factors are very close and indicates there is close to an equal amount of evidence in both models for this rule.

We can make other comparisons showing differences in the two models. For example, there are no rules in the TimberWolf model indicating concepts for cells of different heights or with the same heights. This is because all of the standard cells in the training designs have the same height, so no relationships exist. On the other hand, there are rules in the ArtistII model that consider the height features.

Another comparison shows there is a rule in the ArtistII model that indicates a concept of *way_out* for gate pairs that have the same width. This rule has a certainty factor of 0.670. In the TimberWolf model, there are three rules for cells with the same width, but they indicate three different concepts: *close_2* with a certainty factor of 0.728, *way_out* with a certainty factor of 0.687, and *near_d* with a certainty factor of

0.669. This indicates the ArtistII training set only supports a concept of *way_out* for gates that have the same width. On the other hand, the TimberWolf training set supports three different concepts for cells with the same width. This is a significant difference between the two models.

One more comparison shows another difference between the two models. In the TimberWolf model there is a rule that indicates a relative placement concept of *close_2* for any cell pairs with the same function. This rule has a certainty factor of 0.734. In the ArtistII model, there is a rule that indicates a concept of *way_out* for gate pairs with the same function. This rule has a certainty factor of 0.643. The same feature indicates two different concepts in the two models. The concepts and certainty factors are different because the information in the training sets is different. This along with other comparisons confirms that the learning system did learn different information from the two training sets.

Now that we have compared the TimberWolf model to the ArtistII model, we can show how well the TimberWolf model works on test designs. The test designs on which we used the TimberWolf model include a 4-bit, 8-bit, 16-bit, and 32-bit adder and a 4-bit, 8-bit, and 16-bit multiplier. Because they are test designs, each one has an actual layout from TimberWolf so we can make valid comparisons and derive the appropriate errors.

We applied the TimberWolf model to each of these test designs and produced a list of cell pairs and predicted relative placement concepts for each one. Then we used the estimator to get a preliminary set of estimates. Table 24 shows the area estimates using the TimberWolf model. These height and width estimates are based on an estimated layout using the same "growing" technique as described in Section 4.5. The height and width of the individual cells comes from the cell library. However, we used the same wire area models as described in Section 4.5. It is expected that better wire models will produce better area estimates. The goal for modeling the TimberWolf layout tool was to see if the method could be implemented to handle another layout tool. These results indicate that we are in the ball park for overall error and it is only a matter of tweaking the area models to get better estimates. For example, as the designs get larger the height estimates have more negative error. This indicates the current wire area model does not calculate the required height of

the channels correctly and underestimates it more as the number of cells increases. The width is mostly overestimated because the current model calls for more vertical wire area as a function of the number of rows and cells. In the case of standard cells, existing feedthroughs can be used to route the vertical wires and not as much extra wire area is needed.

Table 24 Height and Width Estimates from TimberWolf Model

Design Name	Number of Cells	Actual Height	Estimated Height	Overall Error	Actual Width	Estimated Width	Overall Error
add4	19	251	274	9.16%	368	367	-0.27%
add8	51	507	538	6.11%	480	516	7.50%
add16	115	687	670	-2.47%	760	830	9.21%
add32	243	1259	1066	-15.33%	1016	1090	7.28%
mult4	79	671	670	-0.15%	536	594	10.82%
mult8	392	1531	1330	-13.13%	1248	1450	16.19%
mult16	1681	4011	2914	-27.35%	2464	3226	30.93%

To estimate delay, we used the same wire delay models as described in Section 4.4. Table 25 shows the delay estimates using the TimberWolf model. These delay estimates are based on the intrinsic delays from the cell library and the wire delay models from before. These delay estimates are well within a 10% overall error and indicate that the wire delay models and the intrinsic delays from the cell library are accurate.

Table 25 Delay Estimates from TimberWolf Model

Design Name	Number of Cells	Actual Delay	Estimated Delay	Overall Error
add4	19	11.2	12.10	8.04%
add8	51	26.5	25.90	-2.26%
add16	115	58.0	56.94	-1.83%
add32	243	122.8	119.02	-3.08%
mult4	79	17.8	17.84	0.22%
mult8	392	43.6	43.34	-0.60%
mult16	1681	102.6	96.14	-6.30%

The main goal here is to show that the modeling method is general enough so it can be used to model other layout tools. We have shown that it is possible, with only minor modifications to the implementation, to model another layout tool. This

section described what modifications were necessary to model the relationships from the TimberWolf layout tool and also showed that we are well within an acceptable range (30% overall error) for the area and well within our target goal (10% overall error) for delay estimates. It is anticipated that minor changes to the wire area and wire delay models in the estimator will produce even better estimates.

5.5 Comparison to Other Area and Delay Estimators

This section shows how our area and delay estimation method compares to other estimators. We compare results from this estimation method to results from the other estimators described in Chapter 2. However, a couple of the estimators in Chapter 2 do not lend themselves to valid comparisons. For example, the Fasolt system [3, 4, 5] is designed for estimating layouts for macrocell based designs only. A comparison between this and our custom gate or even standard cell layout models would be invalid. In addition, Knapp did not compare this layout estimator to other estimation systems. He only showed that his layout estimator could improve initial layouts from 16% to 37% through several reiterations.

The layout area and delay modeling system [23, 22] from Ramachandran et. al. also works on macrocells so a comparison would be invalid here as well. However, they did report an error rate of 13% for area estimates and 10% for delay estimates, based on designs averaging around 2000 cells.

Another estimator, the ADAM system [8, 9], produces a lower-bound area and delay curve which shows different delay values for given area values, all for the same design. To make a valid comparison, we would have to run the ADAM system on one of our test designs to produce this curve. Then we would have to produce several area and delay estimates for the same design using our layout tool model. We can get different area and delay estimates by changing the number of rows in the estimated layout. However, it is impractical to run the ADAM system on our designs. In addition, Jain et. al. compared their lower bound area-delay curves to area-delay estimates based on layouts produced by other systems, but they did not calculate a quantitative error. They could only show that their curve was close and defined a theoretical lower bound for area and delay trade-offs.

We can make valid comparisons with the other layout estimators. Table 26 shows a comparison of results from our area and delay estimation method to results from estimators by Nourani [26] and in the TELE 2.0 [20] and PLEST [7] systems.

Table 26 Comparison to Other Estimators

System Name	Design Name	Dim.	Their Error	Our Error
(Nourani)	mult4	ht	3.09%	0.77%
		wd	4.22%	1.83%
	mult8	ht	4.69%	-1.60%
		wd	6.87%	-15.93%
TELE 2.0	mult16	area	-13.91%	-4.85%
		del	18.00%	-6.30%
	add32	area	9.62%	-9.37%
		del	6.00%	-3.08%
PLEST	mult8	area	5.8%	1.05%
	add16	area	10.6%	6.51%
	mult16	area	3.1%	-4.85%

For example, Nourani and Papachristou [26] used their layout estimator on a 4-bit multiplier design containing 521 transistors. They produced a height estimate that was off by 3.09% from the actual height. They produced a width estimate that was off by 4.22%. Our 4-bit multiplier test design has 550 transistors. Our layout tool model (based on ArtistII) produced a height estimate that was off by only 0.77% and a width estimate that was off by only 1.83%. They also produced estimates for an 8-bit multiplier with 2097 transistors. Their height estimate was off by 4.69% and their width estimate was off by 6.87%. Using the ArtistII layout tool model on our 8-bit multiplier (2574 trans.), the height estimate was off by -1.60% while the width estimate was off by -15.93%. This indicates that we are slightly better than their system or at least very competitive. In addition, this layout estimator only produces area estimates so we could not compare delay estimates.

We can also compare our method to the TELE 2.0 system [19, 20]. The TELE 2.0 system was used on a design containing 1829 cells. This is roughly equivalent to our 16-bit multiplier which has 1681 cells. (The results presented by the TELE 2.0 system are for total area.) For this size of design, their total area estimate was off by -13.91%. Because the TELE 2.0 system uses standard cells, we used results

from our TimberWolf model for comparisons. The TimberWolf model gave us a total area estimate for the 16-bit multiplier that was off by -4.85%. The TELE 2.0 system produced a delay estimate that was off by 18.00% for this design. Our delay estimate was off by only -6.30%.

Another example comparing our layout tool modeling method to the TELE 2.0 system is an adder design. Their adder design contains 225 cells. Our equivalent 32-bit adder design has 243 cells. Their total area estimate is off by 9.62%. Our total area estimate (using the TimberWolf model) is off by -9.37%. Their delay estimate for this design is off by 6.00%. Our delay estimate is off by -3.08%. This comparison and the one before indicate we are very competitive with the TELE 2.0 system. This is even using the TimberWolf model which is not optimized as much as it could be.

The PLEST system [6, 7] is another layout area estimator that we can compare our method against. (This system does not estimate delay.) One of their designs is an 8-bit multiplier. They report a total area estimate that is off by 5.8%. Because this system also uses standard cells, we used our model of TimberWolf. The TimberWolf model produced a total area estimate that was off by only 1.05%. Another design they reported results on is a 16-bit adder. They reported a area estimate that is off by 10.6%. The TimberWolf model produced an area estimate for our 16-bit adder that was only off by 6.51%. They also reported results on a 16-bit multiplier. Their area estimate was off by 3.1%. For our 16-bit multiplier, the area estimate was off by -4.85%.

These comparisons show that our modeling method is very competitive in area and delay estimation. We have also shown that both models, the ArtistII model and the TimberWolf model, can estimate area and delay as well if not better than some of the current estimators.

5.6 Summary

This chapter on experiments addressed several key issues in this research. Through experimentation with a number of training designs and test designs, and two target layout tools, we have demonstrated that our modeling method is valid and quickly produces accurate area and delay estimates. For example, we have shown that the

machine learning method produces better estimates than just randomly assigning relative placement concepts or assigning one concept to all gate pairs. We have also shown that it is possible to produce different models of the same target layout tool using different connfigurations and/or execution parameters for the layout tool. By looking at the modeling error and the overall error, we have shown that we can successfully apply the model to a wide range of design sizes. We also demonstrated that we achieved our goal of less than 30% error with most of the estimates within 10%.

We compared the execution times for pre-processing the target design, applying the model, and estimating area and delay to the time it takes for the target layout tool to just produce a layout. The modeling method is orders of magnitude faster and includes estimating area and critical path delay. Our method is faster even for large complex designs. We also showed good results from applying our model to benchmark designs. Here we showed that our method can also handle large complex sequential circuits as well as combinational circuits. Another set of experiments showed that it is possible to model the input-to-output relationships for other layout tools using the machine learning method with only minor modifications to the implementation. In addition, we showed that the resultant models for the two layout tools we modeled were quite different. Finally, we compared results from both of the models to other area and delay estimators and showed that we are very competitive.

6.0 CONCLUSIONS

This chapter begins by summarizing this thesis. It does this by recapping the motivation for the research, the goals of the research, the solution, and the accomplishments. It also recaps how well the solution met the goals. Next, this chapter discusses the important contributions from this research, what things could have been done differently (hindsight), and the evolution of possible future research.

6.1 Thesis Summary

The research described in this thesis shows it is possible to use machine learning techniques to quickly derive accurate area and delay estimates. Accurate estimates enhance the ability of high-level synthesis systems to produce optimal designs. However, the most accurate estimates are based on information from actual design layouts. These layouts require a long time to produce, so the high-level synthesis system must either use inaccurate estimates or spend a lot of time waiting for a layout. Both ways are ineffective. Therefore, the motivation for this research was to find a way to quickly derive accurate area and delay estimates so the overall effectiveness of high-level synthesis systems could be enhanced. We did this by using machine learning techniques to model the input-to-output relationships that result from using layout tools, rather than modeling the layout itself. We have shown that using the model in place of the layout tool quickly produces accurate area and delay estimates.

In Chapter 2 we looked at several layout estimators and discussed their individual strengths and weaknesses. We also pointed out that each one concentrated on modeling the layout itself, and not the layout tool. Then Chapter 2 described the theory behind layout tools that use a simulated annealing approach. We described the simulated annealing algorithm and looked at two layout tools using this algorithm, ArtistII and TimberWolf. We described these two layout tools because we intended to model each one using the method from this research.

Chapter 2 finished by describing the theory behind machine learning and discussing three common implementations: linear discriminants, neural nets, and production rule based systems. We showed how linear discriminants and neural nets did not satisfy our need to directly check the rules that described the relationships between the inputs and the predicted outputs. Only a production rule based system, such as Rule Learner, produced human readable rules so we could evaluate them during the research. We also described the Rule Learner system in detail and showed that it was very applicable for our research.

In Chapter 3, we described how we came up with the idea of using machine learning to model layout tools. We discussed some alternative approaches such as using "brain-damaged" versions of existing layout tools or using an extensive library of designs and corresponding layouts. We showed how each of these did not meet our goal of quickly deriving accurate estimates. This chapter then described how performing an input-to-output analysis of the target layout tool could lead to a model of the tool. Using this model in place of the actual tool would quickly provide accurate area and delay estimates. Because this approach required a data analysis tool, we looked at different ones and decided on using a machine learning system.

Chapter 3 continued by describing the approach using machine learning to model the input-to-output relationships that result from using layout tools. We introduced a modeling method algorithm and a solution architecture. We described the two main parts of this algorithm: producing the model and applying the model. For each part, we described several procedures from building a training set to deriving area and delay estimates.

Chapter 4 expanded on this algorithm and solution architecture by describing in detail an actual implementation of this method. We used ArtistII as the target layout tool and showed how to build a training set and analyze the training set to produce a layout tool model. We also showed how we can apply this model to test designs to obtain layout information and how this information was used to give us accurate area and delay estimates.

In Chapter 5, we described several experiments that addressed key research issues. First, we showed that it was possible for a machine learning system to learn in

the VLSI CAD domain. We then showed the machine learning method was better than randomly assigning relative placement concepts or using one relative placement concept for all gate pairs. We also showed it was possible to model a target layout tool using different configurations and/or execution parameters for the tool.

Another key issue was determining if the layout tool model could handle a wide variety of design sizes. We showed that it could by applying the model to several designs and demonstrating that both the modeling error and overall error were well within our goals. We showed the area and delay estimates for most of these designs had errors within 10% and none had errors worse than 30%. We also showed that using our model was much faster than producing actual layouts. In fact, we demonstrated that our method could process designs and derive area and delay estimates much quicker than a layout tool could just produce a single layout, even for large complex designs.

We also applied the model to benchmark designs to show that it could handle real world designs, including sequential as well as combinational designs. The area and delay estimates here were also well within our goals.

Chapter 5 went on to describe the generality of the modeling method by using it to model another layout tool, TimberWolf. We not only showed that it was possible to model the input-to-output relationships for another tool, we used this model to produce accurate estimates well within the goals.

Chapter 5 finished by comparing our method to other area and delay estimators. We showed that both models (ArtistII and TimberWolf) are very competitive with current estimators.

6.2 Contributions

The contributions from this research are:

- A method to obtain more accurate area and delay estimates in a reasonably expedient way by modeling the input-to-output relationships that result from using layout tools. Graph application principles are used to obtain accurate area and delay estimates from these layout tool models.

- A generalized and novel method of characterizing layout tools by using an artificial intelligence based machine learning system. This method has demonstrated the ability to work across different design architectures and on different layout tools.

- A new application domain for machine learning systems. This new domain introduces gray areas, i.e., black and white classifications are not necessary. A classification that is "close enough" is sufficient for estimation purposes.

- A set of generalized features that sufficiently describe the target design so relationships between these features and layout concepts can be used in estimating area and delay.

The first contribution aids the design automation process by decreasing the overall design time while increasing the effectiveness of high-level decision making ability. It does this by modeling the input-to-output relationships that result from using the layout tool, and not by analyzing the design itself. This model is used in place of the layout tool to provide the necessary layout information based on the design. This layout information along with graph application principles is used to derive accurate area and delay estimates in a timely fashion. In addition, this method is applicable to a wide range of design sizes (up to several thousand gates) and styles (both combinational and sequential). In fact, we have shown results within our goals for design sizes up to 12,500 gates.

The second contribution is a novel way of analyzing and modeling layout tools. It is a general method using a machine learning system as a data analysis tool and the resultant rules as a representation of the input-to-output relationships that result from using the layout tool. Due to its generality, this method is applicable across different design architectures and may be used to model different layout tools.

The third contribution shows that it is possible to use a machine learning system as a data analysis and modeling tool in a computer-aided design environment. This aids design automation efforts by exhibiting a novel way to analyze and model layout tools and aids artificial intelligence research by opening a new application domain for machine learning. It also shows that exact classifications are not necessary in domains

where reasonable estimates are acceptable. The first set of experiments dealing with learning demonstrates that a machine learning system can function effectively in a computer-aided design environment.

The fourth contribution is a set of features that sufficiently describe the target design. These features are based on graph applications. Some features describe the connectivity, degree, and physical attributes of each node, while the remaining features describe the topology of the design in relation to each node, and relative physical attributes of logically connected node pairs. These features along with layout concepts are important in describing the input-to-output relationships that result from using layout tools. In addition, these features can be generalized when necessary so less information is needed to sufficiently describe the design.

6.3 Hindsight and Evolution

This section suggests future research for the area and delay estimation method by looking back on what could have been done differently and looking forward to new directions.

In the area of learning, one different tactic is to use neural nets instead of a production rule system. As described in Section 2.3.3, we used a production rule learning system so we could evaluate the rules directly. In some cases, it was also necessary to compare different sets of rules to see how they were different or similar. However, after we were confident in the learning procedure and had a quality suite of training sets, we could use a neural network to learn the necessary concepts. This would speed up the learning process and the model application process. However, it may prove difficult to incorporate physical limitations in the application of a neural net model. We have no certainty factors to rank first and second choice concepts. One possible solution is to incorporate the physical limitations into the training sets. This would increase the complexity of the learning process, but may yield better model results.

Another different tactic is to model ArtistII and ClusterII together. This means using ClusterII to provide an initial clustering so ArtistII can yield better layouts. This would be helpful when dealing with large training designs. We could get better

layouts from which to learn. This also gives chip designers more flexibility because they can choose between different models: some based on ArtistII alone and some based on ArtistII and ClusterII together.

In the area of estimation, we can do a number of things differently. One improvement is to check for false paths in the delay estimator. (In this research, we verified the estimated critical paths of all the test designs against the actual critical paths found by IRSIM.) It is possible for a target design to have the estimated worse case path turn out to be unachievable for any input data vector. An estimator that checks for this would enhance a chip designer's confidence in the estimated delay.

Another estimation improvement is to use better wire area models. In this research we used a very simple estimation for wire area. More columns requires more horizontal wires and more rows requires more vertical wires. A better method might be to use Rent's rule [21] or the Left Edge Algorithm [59]. However, this would require a better placement technique. In this research, we placed nodes into an estimated layout so they would satisfy the relative distance between node pairs. We gave no consideration to interconnections among more than two nodes. Using Rent's rule or the left edge algorithm produced overly large wire areas. Perhaps learning the concept of net types would lead to better placement. Instead of placing two nodes according to their predicted relative distance, we would place all nodes from a single net into the estimated layout. This would be done one net at a time. Then maybe Rent's rule and the left edge algorithm would yield better wire areas.

In the area of using the model, one suggestion is to completely automate the process. The user would have a set of models to choose from and would only have to enter a high-level description of the target design into the system. This high-level description would be automatically compiled and translated into the proper format. The user selected model would be applied and area and delay estimates would be produced. Automating the process would significantly increase the speed and reduce errors from human mistakes. This in turn would increase the overall effectiveness of high-level synthesis systems using our method.

APPENDIX A

APPENDIX A

The following is a list of definitions used in this research:

Architecture: refers to the instruction set, physical components and timing to which all hardware implementations must adhere [71]. The architecture of a given design transcends all three abstraction levels: behavioral, structural, and physical.

Implementation: refers to specific hardware designs using a particular architecture [71].

Design Automation: refers to using computer-based routines to perform the design process [72].

Design Automation Tools: computer-based software routines that can handle designs that are too large or complex to undertake otherwise. Also provides for shorter design time, improved product quality (performance and reliability), and reduced product costs [72].

VLSI CAD Design Process: refers to a sequence of transformations on design representations at the three levels of abstraction - behavioral, structural, and physical [72].

High Level Synthesis: refers to the process where an abstract behavioral specification of a digital system is transformed into a register transfer level structure that realizes the given behavior [1].

Behavior: refers to the way the system or its components interact with their environment; the mapping from inputs to outputs [1].

Structure: refers to the set of interconnected components that make up the system [1].

Physical Design: refers to the process of reducing a structural description of a given design down to the geometric layout of an integrated circuit [73].

Physical Layout: refers to a set of planar geometric shapes in several layers that represent the low level components and interconnections of a given design [74].

Placement: refers to the problem of assigning locations to fixed blocks that represent components on a layout surface [74]. Relative placement refers to the placement of two components relative to each other.

Floorplanning: refers to a placement problem where the blocks that represent components are not fixed [74].

Machine Learning: refers to using a computer program to acquire knowledge [50]. May also refer to using a computer program as a data-analysis tool for scientists and other investigators [54].

Inductive Learning: refers to the inference of a generalized conclusion from particular instances provided by a teacher or environment [50].

Features: refers to a set of attribute-value pairs that describe the characteristics of the domain to be learned [53].

Training instance: refers to a set of features that describe one example from the domain to be learned [53]. Each training instance has the same set of attributes describing the features, but with different values depending on the characteristics of that instance.

Training set: refers to a set of training instances [53].

Concept: refers to an abstract or generic classification generalized from particular instances [50].

Data: refers to words and or numbers which have specific meaning. Data derive their meaning from a precise name and agreed upon definition [75].

Information: refers to data that have been organized and arranged to convey knowledge [75].

Method: refers to a means or manner of procedure, a regular and systematic way of accomplishing something, an orderly and systematic arrangement. Also refers to a set of procedures that follow a detailed, logically ordered plan [75].

Methodology: refers to a system of principles, practices, and procedures applied to a specific branch of knowledge [75].

APPENDIX B

APPENDIX B

This technical report describes how to use the machine learning method of modeling the input-to-output relationships that result from using a layout tool. It also describes how to use this model (a set of rules) in place of the layout tool to quickly obtain accurate area and delay estimates. This report follows an algorithm that describes how these relationships are modeled and how the model is used to produce area and delay estimates. This report describes each step of the algorithm and to clarify the discussion, uses an example of modeling an actual layout tool and using the model to estimate area and delay.

A machine learning tool called **Rule_Learner** learns the input-to-output relationships and produces the set of rules. Another tool called **Match** (written exclusively for this application) formats the training and target designs and estimates area and delay. In the estimator mode, the Match tool calls another tool, **IE** (Inference Engine), to apply the rules to a target design and produce a list of predicted layout concepts. The Match tool uses this list to estimate area and delay. The target layout tool is **ArtistII** from the **Keystone Tool Suite**. The operations of each of these tools are fully described in the appendix. The complete solution architecture as implemented for modeling ArtistII using these tools is shown in Figure 44.

All of the tools and example files described by this report are located on the machine jaguar.ee.pitt.edu, under the directory /usr/users/software/est. The subdirectory location of the tools and example files are given in this report as they are described. To access the tools described in this report, the user should add the following line to their path environment:

```
set path = ($path /usr/users/software/est/bin)
```

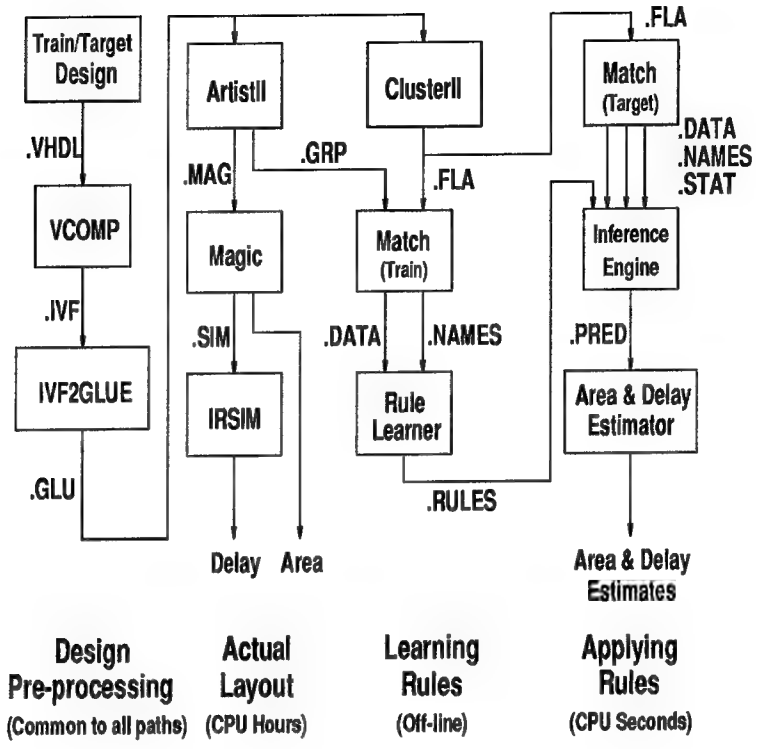


Figure 44 Solution Architecture

Modeling Method Algorithm

There are two parts to the algorithm: producing the model and using the model. Producing the model uses the Rule_Learner tool as a data analysis tool to search through a set of training data looking for trends and characteristics that determine the relationships from using the target layout tool. Using the model involves applying the model to a target design to get predicted layout information and then using this information to derive area and delay estimates.

Part One: Producing the Model

The first part of the algorithm consists of two procedures: building a training set and analyzing the training set.

Procedure One: Building a Training Set

In this procedure, the user selects a set of training designs and processes them through the target layout tool into actual layouts. Information from the training designs and their corresponding layouts is processed into a training set. This training set represents the trends and characteristics that the target layout tool exhibits on the training designs.

Step 1. The first step selects a set of training designs. These designs determine the characteristics of the layout tool being modeled. For this report, the example training set is made up of five designs (*4-bit adder*, *8-bit adder*, *12-bit adder*, *16-bit adder* and *32-bit adder*).

Step 2. The second step processes the training designs into a format the target layout tool can handle. For this application, this is done by several tools from the high level synthesis system that uses ArtistII: the Keystone Tool Suite.

Because ArtistII requires representation of designs in the GLUE format, the training designs, written in VHDL, are translated into the GLUE format by first compiling them using the tool called VCOMP. VCOMP compiles the designs into the Intermediate VHDL Format (IVF). Another tool, IVF2GLUE, is used to translate this IVF format into the GLUE format. In addition, the formatting tool, Match, requires a flattened GLUE file. The tool called ClusterII flattens the GLUE file. For convenience, the example training designs have already been processed by these tools into a flattened GLUE format. These files have a *.fla* extension (i.e., *add4.fla*) and are stored in the directory /usr/users/software/est/learn/train.

Step 3. The third step processes each training design through the target layout tool into an actual layout. If the target layout tool is reconfigurable and/or has user set parameters, these should be set to the desired values that the model is to reflect. A training set and the resultant model can only reflect one configuration of the layout tool using one set of parameter values.

For the example, ArtistII processed each training design into an actual layout using 10,000 iterations. The layout positions for the nodes in the training designs are listed in the files with a *.grp* extension (i.e., *add4.grp*) in directory /usr/users/software/est/learn/train.

Step 4. The fourth step involves processing the training designs and their layout information into a training set data format. The Match tool formats the training designs based on format requirements from the Rule_Learner tool.

To run the example, copy all of the files from /usr/users/software/est/learn/train to a working directory. Then, to process a training design into the proper format, use the match tool. For example, to process the 4-bit adder, issue the command:

```
match -t add4
```

The -t option puts the Match tool into training mode (see the man page for match). The Match tool parses the files *add4.fla* and *add4.grp* to get the relevant design and corresponding layout information. Match also obtains physical information about the nodes in the training designs by parsing the file called *anodes.phy.* Match then produces a training data file called *add4.data* and a domain description file called *add4.names.* In addition, Match's output to the screen should appear like this:

```
Filename: add4
Begin processing flattened description file.
Begin processing layout description file.
Layout: 8 rows by 8 cols.  Number of nodes: 43.
Generate adjacency matrix.
Transform distance matrices.
Produce data files.
```

Repeat this step for each of the remaining training designs: *add8, add12, add16,* and *add32.* When all five training design files have been created, concantenate them into one training set file. To do this, issue the command:

```
cat add4.data add8.data add12.data add16.data add32.data > adder.data
```

This will produce the complete training set in the file *adder.data*. If this file is the same as the file *example.data* in the directory /usr/users/software/est/learn/train, then the process was properly executed. If they are not the same, the individual training data files should be deleted and this step repeated. In addition, a domain description file must be created. This can be done by copying any one of the training design domain description files (i.e., *add4.names*) to *adder.names*.

Procedure Two: Analyzing the Training Set

The Rule_Learner tool analyzes the training set for relationships between design features and layout concepts. A set of rules represents any relationships found. These relationships indicate trends and characteristics of the target layout tool.

Rule_Learner searches through the training set data looking for evidence that supports relationships between the design features and the relative placement concepts. If the confidence factor for a particular relationship meets or exceeds a user-set threshold, a rule describing that relationship is produced. The threshold is determined by a parameter set by the user before executing the learning system.

This procedure requires the files *adder.data* and *adder.names* produced by the Match tool earlier. The training set data is stored in the file *adder.data* and the domain description data is stored in the file *adder.names*. To learn the rules that represent the relationships based on these files, issue the command:

```
rule_learner adder 0.9 0.6 0.1 6 50
```

As described in the man page for *rule_learner*, the bias parameters listed after the file name tell the Rule_Learner tool to start searching for rules with certainty factors greater than or equal than 0.9, and to repeat the search for certainty factors greater than or equal to 0.8, 0.7, and 0.6 (decrement the threshold by 0.1 for each repetition). Therefore, Rule_Learner will perform a total of four searches. The remaining bias parameters tell Rule_Learner to allow no more than 6 conjuncts in the left-hand side of the rules generated and to use a beam width of 50. The rules produced by Rule_Learner will be listed in the file *adder.rules*. This file should be the same as

the file *example.rules* from the directory /usr/users/software/est/learn. (Due to its length, the screen output from using Rule_Learner in this example is not shown here. However, it is stored in the file *example.drib* from /usr/users/software/est/learn. If the screen output from Rule_Learner is redirected to a file, it should look the same as *example.drib*.)

Part Two: Using the Model

The set of rules produced by Rule_Learner makes up the model of the relationships from using the target layout tool. To apply the model, the target design must be pre-processed in the same fashion as a training design. However, a layout is not produced. Instead, the model determines what layout concepts result from certain features in the design. This information is provided to the Match tool so it can derive area and delay estimates.

Procedure Three: Applying the Model

There are three steps in this procedure. The Match tool pre-processes the target design, calls IE to apply the model (set of rules), and uses the resultant list of predicted layout concepts to estimate both area and delay.

Step 1. The first step pre-processes the target design into the proper format. This format is identical to that for a training design, but because no layout is produced, none of the layout concepts are included. Pre-processing the target design into this format makes it straightforward to match features from the target design to those in each of the rules.

The example uses a 16-bit adder as an target design. The flattened GLUE description of this design (*add16.fla*) is located in directory /usr/users/software/est/target. To perform this procedure, copy this file to the working directory. To pre-process, apply rules to, and estimate area and delay for the 16-bit adder, issue the following command:

```
match -sadder -r8 add16
```

(As described in the man page for match, this particular command puts Match into the estimator mode so it will pre-process the 16-bit adder, call IE to apply the rules from the file *adder.rules*, and estimate area and delay based on an estimated layout using 8 rows.)

To pre-process this target design, The Match tool parses the file *add16.fla* and produces a file called *add16.data*. Match also produces a file called *add16.names*. The *add16.data* file contains the data against which the rules are applied. The *add16.names* file contains the domain description. This file provides a context in which the rules are applied.

Step 2. In the second step, the Match tool automatically calls the IE tool to apply the rules. In the example, the rules come from the file *adder.rules*. In this file, the rules are listed in descending order according to their certainty factors. The IE tool searches the target design data first using the rule with the highest certainty factor. The data is searched again using the rule with the next highest certainty factor, and so on. After all data has been searched using all rules, the IE tool produces a file containing the predicted layout concepts. For the example, this file is called *adder.pred*.

Step 3. In the third step, the Match tool uses the list of predicted layout concepts from the file *adder.pred* to estimate area and delay. To estimate area, the Match tool produces an estimated layout based on the predicted layout concepts. In the example, the Match tool produces an estimated layout using 8 rows in the file *add16.clu*. The estimated layout file looks like this:

```
(test.clu
(row r1 nil nil g122 nil nil nil nil nil g194 g70 g95 g176 g175 g94 g184 g99
g78 g167 g166 g47 g141 g96 g150 g162 g105 g66 g50 g163 g108 g103 g72 g104)
(row r2 g185 g112 g158 g121 g157 g149 g148 g67 g68 g97 g140 g139 g86 g193 g106
g123 g87 g186 g132 g131 g136 g190 g197 g64 g51 g154 g159 g109 g55 g49 g172 g189)
(row r3 g115 g130 g76 g144 g124 g160 g133 g153 g88 g120 g81 g77 g142 g113 g85
g69 g118 g54 g48 g181 g117 g58 g52 g91 g145 g100 g101 g56 g198 g92 g53 g74)
(row r4 g138 g129 g151 g79 g75 g171 g83 g180 g90 g80 g164 g155 g146 g135 g126
g114 g61 g60 g73 g127 g59 g82 g137 g128 g119 g57 g110 g89 g191 g182 g173 g71)
(row r5 nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil g188
g179 g170 g1 g161 g152 g143 g2 g134 g125 g116 g3 g107 g98 g4 g46)
(row r6 nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil g5 g45
g43 g42 g6 g41 g40 g39 g7 g38 g37 g36 g8 g35 g34 g33)
(row r7 nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil g9 g32
```

```
g31 g30 g10 g29 g28 g27 g11 g26 g25 g24 g12 g23 g22 g21)
(row r8 nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil g13
g20 g19 g18 g14 g17 g16 g15 g0 g196 g195 g187 g178 g177 g169 g168)
(column a1 r1 r2 r3 r4 r5 r6 r7 r8)
)
```

The purpose of this estimated layout is to provide information for area estimation. It should not be misconstrued as a substitute for an actual layout for any other purpose.

Next, the Match tool uses the predicted layout concepts to find the wire delay for all interconnections in the target design. This information, along with individual node delay information from the file *anodes.phy*, is used by Match to determine the critical path and the critical path delay. In the example, the critical path including delay information is listed in the file *add16.crit*. (This file is too large to reproduce here. However, the format is identical to output from the IRSIM simulation tool.)

When the Match tool is finished, the file *add16.clu* should be the same as *example.clu* and the file *add16.crit* should be the same as *example.crit*. (These example files are located in the directory /usr/users/software/est/target for comparison purposes.) In addition, the screen output from Match should look like this:

```
Filename: add16
Begin processing flattened description file.
Layout: 8 rows by 32 cols.  Number of nodes: 199.
Generate adjacency matrix.
Transform distance matrices.
Produce data files.

Apply layout tool model (set of rules):
Read 256 cases (11 attributes) from adder.test
Read 31 rules from adder.rules

Find height and width estimates:
Estimated Height = 944 lambda.
Estimated Width  = 1492 lambda.
```

```
Find critical path delay.
Predicted Delay[0] = 58.92 ns.
Predicted Delay[1] = 90.37 ns.
```

Other Design Examples

This section gives the locations and names of other designs that were used to validate the machine learning method. The general procedures described above may be repeated for any and all of these designs. These designs may also be used as either training designs or target designs.

These designs are located in the directory /usr/users/software/est/test. There are 24 test designs, the same ones used in validating the method. Each design is represented by a flattened GLUE description (*file.fla*). Each design also has a corresponding layout description (*file.grp*) so any design may be used as a training design as well as a target design. Each of these layout descriptions (except for the 16-bit multiplier) were produced by ArtistII using 10,000 iterations. The 16-bit multiplier layout description (*mult16.grp*) was produced by ArtistII using only 100 iterations. For completeness, an alternative set of training designs with layouts from ArtistII using only 100 iterations are stored in the directory /usr/users/software/est/test/alt. These may be built into a training set using the above procedures. The resultant set of rules would model the input-to-output relationships that result from using ArtistII with only 100 iterations.

Summary

This technical report describes how to use the machine learning method of modeling input-to-output relationships that result from using layout tools. It also describes how to use this model to estimate area and delay. The example given here was based on modeling the relationships from using the layout tool ArtistII. It was also based on using the machine learning tool Rule_Learner. Another tool called Match was used to format the flattened GLUE descriptions of the training and target designs into the

appropriate data and domain description files, apply the model (set of rules), and estimate area and delay. The Match tool called another tool, IE, to actually apply the rules and produce a list of predicted layout concepts. The Match tool then used the information in this file to estimate the area and delay.

The example described here was also suplemented by example files for comparison purposes. The same procedure and tools may be used to build other training sets, produce different models of ArtistII, and use these models to estimate area and delay for other target designs.

APPENDIX C

APPENDIX C

NAME

artistII - a CMOS, two dimensional, gate matrix module generator

SYNOPSIS

artistII -ln -rn -gn -an -e -f -mcell -dn -q file

DESCRIPTION

ArtistII is a CAD tool which does transistor placement and routing for double layer metal, two dimensional, CMOS gate matrix layouts using cluster based simulated annealing. It works best for small and medium size modules (fewer than 100 gates). Larger modules require longer running times for satisfactory results. ArtistII uses as its input file a GLUE description of the circuit (file.glu) and, optionally, a cluster description (file.clu). It outputs a Magic format layout file (file.mag) describing the smallest two dimensional CMOS gate matrix layout it found and a cluster grouping description file (file.grp) which lists the clusters used in the that layout.

Arguments are:

-ln This optional argument indicates the number of trial placements to use. The larger the number of trial placements the smaller the final layout, but the longer the running time. Defaults to "1".

-rn This optional argument indicates the number of well sharing folds to make. Defaults to "1".

-gn This optional argument indicates the number of layout permutations to make between evaluations as a percentage of the maximum number of possible permutations. Defaults to 0%; i.e., only one layout permutation is made between evaluations. A larger percentage will speed up the search process, but too large a percentage may degrade the layout results. Values around 2% seem to work well.

-an This optional argument specifies the search strategy to use. The values for n range from 0 (ramp accept function with zap back strategy) to 5. Defaults to "0". It is recommended that the default be used for the best layout results.

-e This optional argument tells artistII to just evaluate the layout without creating the .mag file for the layout. This will speed up the program considerably, but cannot be used to generate layouts, just layout statistics. Defaults to "off".

-f This optional argument tells artistII to produce a flattened GLUE description in .fla. This description is annotated with information about the row/column placement of each gate. By renaming the .fla file as the .glu file and the .grp file as the .clu file, artist can, in one iteration, reproduce the layout.

-mcell

This optional argument allows the designer to tell artistII which cell to layout when the input GLUE file contains more than one topmost cell. Alternatively, if the module to be laid out is not the topmost cell it can be designated using this option.

-dn This optional argument turns on the debug flag. Various values of n will give various debugging information - "0" gives no diagnostic information, "1" records the status of only the "best" iterations, "2" records the status of the other iterations, "4" lists the final gate ordering by column for the layout in a GLUE like format. Defaults to "0".

-q This optional argument causes artistII to run in quiet mode so that the GLUE listing is suppressed.

EXAMPLE

To use artistII to produce the layout for the carry circuit of a binary full adder (stored in carry.glu) issue the command

```
artistII -l5 -r1 -d7 carry
```

and artistII will generate the following diagnostics as well as creating the carry.mag file.

```
** ArtistII - Version 2.03 -
Last updated  7/17/90 at 17:08:55 **
```

```
Options: module(carry) loop(5) row(1) debug(0x7)
accept(ramp)

***************** Module Description ********************

***     1 ***  carry(a, b, cin, cout)
***     2 ***  {
***     3 ***       _oai21(a, b, cin, t0);
***     4 ***       _ai2(a, b, t1);
***     5 ***       _ai2(t0, t1, cout);
***     6 ***  }
***     7 ***
***     8 ***  _ai2(i0[], i1[], o[])
***     9 ***  {
***    10 ***       <i0[] & i1[]> o[] = 0;
***    11 ***       <!i0[] | !i1[]> o[] = 1;
***    12 ***  }
***    13 ***
***    14 ***  _oai21(i0[], i1[], i2[], o[])
***    15 ***  {
***    16 ***       <(i0[] | i1[]) & i2[]> o[] = 0;
***    17 ***       <(!i0[] & !i1[]) | !i2[]> o[] = 1;
***    18 ***  }

Can't open carry.clu - dumb mode
numstm -> 3 numrow -> 1 numopr -> 2 numvgd -> 4

num  size  best  min  row  col    lev    acp    grp
  0   120    -1   -1   10   12  0.000  0.000  0.000
best - accepted
weights tre ->   4 grd ->** opr ->** T
  1   120   120  120   10   12  0.000  0.000  0.000
```

```
same - accepted
weights tre ->   4 grd ->** opr ->** T
  2   108   120  120    9   12  0.000  0.000  0.000
best - accepted
weights tre ->   4 grd ->** opr ->** T
  3   120   108  108   10   12  0.000  0.000  0.000
worse - rejected
weights tre ->** cmp grd ->   3 opr ->   2 G
  4   120   108  108   10   12  0.000  0.000  0.000
worse - rejected
weights tre ->** cmp grd ->   3 opr ->   2 G
  5   120   108  108   10   12  0.000  0.000  0.000
worse - rejected

best layout -> 108

carry_0(cout, cin, b, a)
{
     <!a % !b> 1 = t1;
     <!a # !b> 0 = t1;

     <!t1 % !t0> cout = 1;
     <!t1 # !t0> cout = 0;

     <(!a # !b) | !cin> 1 = t0;
     <(!a % !b) & !cin> 0 = t0;
}

********** Statistics **********

memory used 40 kbytes
input time used 0 seconds user 0 seconds system
```

compute time used 0 seconds user 0 seconds system

The search strategy used in the example above is simulated annealing with zap back using a ramp accept function. With a ramp accept function, smaller layouts are accepted with some increasing probability, while larger layouts are accepted with some decreasing probability. Other accept functions are greedy, cut-off, and step. When a zap back occurs the "best" layout found so far (i.e., the smallest layout measured in terms of the number of interconnect rows times the number of interconnect columns) is used as the starting layout for the next annealing move. The next layout may be "best" layout so far (which becomes the zap back target), "better" than the preceding one (which may be rejected early in the annealing process, accepted later in the annealing process, or replaced by the zap back target), the "same" size as the preceding one (which may be accepted, rejected, or zapped), or may be "worse" than the previous one (which may be accepted early in the annealing process, rejected later in the annealing process, or zapped). With fewer than 100 iterations a greedy accept function is actually used (i.e., only best, better, and same are accepted) and with fewer than 1000 iterations no zap back occurs. Theoretically, a simulated annealing algorithm will converge to the global minimum if run for a sufficient number of trial placements. For this simple circuit artistII found the best layout on the second trial placement, one of size 108 requiring 9 layout rows and 12 layout columns.

In this example artistII runs in "dumb mode" since no cluster file (carry.clu) exists. In dumb mode, a randomly generated cluster tree is used to construct annealing moves.

In smart mode, the annealing process moves gate clusters as specified in the .clu file. A .clu file can be created with clusterII. The "weights" diagnostics provide information about the number and type of layout permutations allowed at each iteration: "tre" is the number of cluster permutations, "grd" is the number of intra-cluster permutations, and "opr" is the number of intergate (i.e., transistors within a gate) permutations. A "**" entry indicates that that type of move is not considered at that iteration. A trailing T, G, or O (tre, grd, opr) indicates the actual permutation used.

The layout style used by artistII is two dimensional CMOS gate matrix. A two dimensional gate matrix extends the conventional (one dimensional) gate matrix layout to include more than one pull-up/pull-down row with routing through rows and routing through gates within a row. The GLUE-like listing of the circuit description given after the annealing diagnostics not only gives the gate ordering in the final gate matrix layout by interconnect column, but also provides information about the transistor ordering by interconnect row. If a "#" ("%") is used in place of a "&" ("|") it means that that and (or) operator is commutative and the transistors can be placed in any order in the transistor vertical (horizontal) chain. Otherwise, the gate ordering is from left to right working from the Vdd line for pull-ups and the GND line for pull-downs. Layout produced by artistII conform to MOSIS Rev 6 design rules which are valid for lambda's of 1.5, 1.0, 0.8, and 0.6. ArtistII uses fixed size transistors (2 lambda by 4 lambda for nfets and 2 lambda by 6 lambda for pfets) in the absence of annotation information in the GLUE file, Vdd and GND lines of 6 lambda, and two layer metal. Stacked via's are not supported.

Plugs are inserted at least one set per gate pair at the well boundaries. The plot which is 112 lambda wide by 96 lambda wide corresponding to the carry.mag file is shown below. Note that the gate and transistor ordering is that specified in the GLUE like output given as part of the the diagnostics.

FILES

~cad/psutools/bin

SEE ALSO

clusterII(1), ivf2glue(1), sim2glue(1), glue(5)

AUTHORS

Robert M. Owens, F-S Fuh, P-P Hou, K. Pun

APPENDIX D

APPENDIX D

```
NAME
  rule_learner - a knowledge-based, production rule machine
  learning system.

SYNOPSIS
  rule_learner [ file <bias-parameters> ]

DESCRIPTION
  The domain description (attribute/value/class information)
  and training set data are stored in files called file.names
  and file.data, respectively. The format for these files is
  the same as for the C4.5 learning system (by Quinlan).

  <bias-paramenters>:

    start-cf-thresh stop-cf-thresh stepsize MaxConjs BeamWidth

  Bias Parameters:

       start-cf-thresh
         starting threshold for minimum certainty factor of
         a satisfactory rule

       stop-cf-thresh
         stopping threshold for minimum certainty factor of
         a satisfactory rule

       stepsize
```

decrement value for going from starting certainty factor threshold to stopping certainty factor threshold

MaxConjs
maximum number of conjuncts allowed in rule antecedent

BeamWidth
width of beam in beam search

rule_learner begins by searching through the rule space (defined by the training set data and domain description) for rules with certainty factors greater than or equal to start-cf-thresh. After all rules have been found, it decrements start-cf-thresh by stepsize and repeats the search. rule_learner implements a simple inductive strengthening heuristic: learn a new rule only if it covers an example not covered by any previously learned rule. The searches are repeated until stop-cf-thresh is reached. The last search uses stop-cf-thresh as the minimum certainty factor of a satisfactory rule. The satisfactory rules found are printed out (in a human readable format) at the end of the run.

rule_learner also saves the rules learned (in a non-human readable format) to file.rules. This file is used by the inference engine tool (ie) when applying the rules.

EXAMPLE

This example comes from a computer aided design (CAD) domain application where rule_learner is used to model the input-to-output relationships that result from using layout

tools. The training set is made up of information from training designs and their corresponding layouts. The layouts were produced by the layout tool artistII. The training set data were produced by a program called match. For this example, the training set data is stored in the file adder.data and the domain description data is stored in the file adder.names. To learn the rules that represent the relationships based on these files, issue the command:

```
rule_learner adder 0.9 0.6 0.1 6 50
```

The bias parameters in this example tell rule_learner to start searchin for rules with certainty factors greater than or equal to 0.9, and to repeat the search for certainty factors greater than or equal to 0.8, 0.7, and 0.6 (decrement the threshold by 0.1 for each repetition). Therefore, rule_learner will perform a total of four searches. The remaining bias parameters tell rule_learner to allow no more than 6 conjuncts in the left-hand side of the rules generated and to use a beam width of 50.

rule_learner will output the following information to the screen as well as saving the rules learned (adder.rules):

```
Read 1144 cases (11 attributes) from adder.data

 Current Bias:  Search_type: cf-based cf thresh= 0.900
 MaxConjs= 6 BeamWidth= 50

 Ready to sort.  Queue length is 7

Exs to cover: 1144
 Ready to sort.  Queue length is 168
```

```
 500
Exs to cover: 1144
 Ready to sort.  Queue length is 518 1000
Exs to cover: 1144
 Ready to sort.  Queue length is 330

Exs to cover: 1144
 Ready to sort.  Queue length is 244

Exs to cover: 1144
 Ready to sort.  Queue length is 196 1500
Exs to cover: 1144
 Ready to sort.  Queue length is 144

Exs to cover: 1144

 Found 0 Good Rules (before filtering)

    0 examples covered (out of 1144)

 Found 0 Good Rules

 POLICY: 0 new good rules
 score of 0 new good rules alone: 0.000000
 score of 0 old good rules alone: 0.000000
 score of 0 old good rules in new_all: 0.000000
 POLICY: 0 rules before filtering
 POLICY: 0 rules after filtering
***
Read 1144 cases (11 attributes) from adder.data

Exs-to-cover:  1144
```

```
 POLICY: 1144 examples to cover

 Current Bias:  Search_type: cf-based cf thresh= 0.800
 MaxConjs= 6 BeamWidth= 50

 Ready to sort.  Queue length is 7

Exs to cover: 1144
 Ready to sort.  Queue length is 168
 2000
Exs to cover: 1144
 Ready to sort.  Queue length is 518
 2500
Exs to cover: 1144
 Ready to sort.  Queue length is 330

Exs to cover: 1144
 Ready to sort.  Queue length is 244
 3000
Exs to cover: 1144
 Ready to sort.  Queue length is 196

Exs to cover: 1144
 Ready to sort.  Queue length is 144

Exs to cover: 1144

 Found 0 Good Rules (before filtering)

    0 examples covered (out of 1144)

 Found 0 Good Rules
```

```
 POLICY: 0 new good rules
 score of 0 new good rules alone: 0.000000
 score of 0 old good rules alone: 0.000000
 score of 0 old good rules in new_all: 0.000000
 POLICY: 0 rules before filtering
 POLICY: 0 rules after filtering
***
Read 1144 cases (11 attributes) from adder.data

Exs-to-cover:  1144
 POLICY: 1144 examples to cover

 Current Bias:
 Search_type: cf-based
 cf thresh= 0.700  MaxConjs= 6   BeamWidth= 50

 Ready to sort.  Queue length is 7

Exs to cover: 1094
 Ready to sort.  Queue length is 167
 3500
 beam before: 516
 beam after: 351
Exs to cover: 1036
 Ready to sort.  Queue length is 341
 4000
 beam before: 280
 beam after: 166
Exs to cover: 983
 Ready to sort.  Queue length is 143
```

```
 beam before: 186
 beam after: 102
Exs to cover: 983
 Ready to sort.  Queue length is 93
 4500
 beam before: 154
 beam after: 77
Exs to cover: 983
 Ready to sort.  Queue length is 77

 beam before: 110
 beam after: 55
Exs to cover: 983
 Ready to sort.  Queue length is 55

 beam before: 0
 beam after: 0
Exs to cover: 983

 Found 43 Good Rules (before filtering)

    161 examples covered (out of 1144)

 Found 15 Good Rules

 POLICY: 15 new good rules
 score of 15 new good rules alone: -45363.078125
 score of 0 old good rules alone: 0.000000
 score of 0 old good rules in new_all: 0.000000
 POLICY: 15 rules before filtering
 POLICY: 15 rules after filtering
***
```

```
Read 1144 cases (11 attributes) from adder.data

Exs-to-cover:  1094    1038    1036    1036    1036
1036    1036    1036    1036    1036    1036    1017
983    983    983
 POLICY: 983 examples to cover

 Current Bias:
 Search_type: cf-based
 cf thresh= 0.600  MaxConjs= 6   BeamWidth= 50

 Ready to sort.  Queue length is 7

 beam before: 168
 beam after: 147
Exs to cover: 694
 Ready to sort.  Queue length is 139
 5000

 beam before: 556
 beam after: 396
Exs to cover: 569
 Ready to sort.  Queue length is 376
 5500
 beam before: 344
 beam after: 214
Exs to cover: 376
 Ready to sort.  Queue length is 193

 beam before: 104
 beam after: 53
Exs to cover: 376
```

```
 Ready to sort.  Queue length is 53

 beam before: 32
 beam after: 16
Exs to cover: 376
 Ready to sort.  Queue length is 16

 beam before: 0
 beam after: 0
Exs to cover: 376

 Found 49 Good Rules (before filtering)

    768 examples covered (out of 1144)

 Found 31 Good Rules

 POLICY: 31 new good rules
 score of 31 new good rules alone: -10815.149414
 score of 15 old good rules alone: -45363.078125
 score of 15 old good rules in new_all: -45363.078125
 POLICY: 46 rules before filtering
 POLICY: 31 rules after filtering
***
Read 1144 cases (11 attributes) from adder.data

Exs-to-cover: 1085   1059   1059   1034   1034
727    727    727    687    687    638    638    638
638    638    636    634    578    528    528    528
472    472    429    376    376    376    376    376
376    376
 POLICY: 376 examples to cover
```

```
Rules:

(nt1_nt2 !=) (nx1_nx2 !=) (ni1_ni2 !=)  ==> close_2
 { pos=  0.46,  neg=  0.28,  cf=  0.622
  p=59, n=281, tp=127, tn=1017 }

(com_src no) (a1_a2 !=) (ni1_ni2 !=)  ==> near_s
 { pos=  0.32,  neg=  0.17,  cf=  0.636
  p=26, n=185, tp=81, tn=1063 }

(h1_h2 !=) (nx1_nx2 !=) (ni1_ni2 !=)  ==> close_2
 { pos=  0.46,  neg=  0.28,  cf=  0.622
  p=59, n=281, tp=127, tn=1017 }

(com_src no) (a1_a2 !=) (ni1_ni2 !=)  ==> close_2
 { pos=  0.29,  neg=  0.17,  cf=  0.622
  p=37, n=174, tp=127, tn=1017 }

(w1_w2 !=) (nx1_nx2 !=) (ni1_ni2 !=)  ==> close_2
 { pos=  0.46,  neg=  0.28,  cf=  0.622
  p=59, n=281, tp=127, tn=1017 }

(a1_a2 ==) (nx1_nx2 ==)  ==> way_out
 { pos=  0.50,  neg=  0.29,  cf=  0.630
   p=307, n=152, tp=619, tn=525 }

  (com_src no) (a1_a2 !=) (nx1_nx2 ==)  ==> close_2
   { pos=  0.20,  neg=  0.10,  cf=  0.645
    p=25, n=104, tp=127, tn=1017 }

  (com_dest no) (nx1_nx2 !=) (ni1_ni2 !=)  ==> close_2
```

```
 { pos=  0.46,  neg=  0.28,  cf=  0.618
  p=59, n=286, tp=127, tn=1017 }

(com_src no) (h1_h2 !=) (ni1_ni2 !=)  ==> near_s
 { pos=  0.81,  neg=  0.51,  cf=  0.613
  p=66, n=537, tp=81, tn=1063 }

(com_src no) (nt1_nt2 !=) (ni1_ni2 !=)  ==> near_s
 { pos=  0.81,  neg=  0.51,  cf=  0.613
  p=66, n=537, tp=81, tn=1063 }

(w1_w2 !=) (nx1_nx2 ==)  ==> far_1
 { pos=  0.47,  neg=  0.28,  cf=  0.621
  p=49, n=291, tp=104, tn=1040 }

(h1_h2 !=) (nx1_nx2 ==)  ==> far_1
 { pos=  0.47,  neg=  0.28,  cf=  0.621
  p=49, n=291, tp=104, tn=1040 }

(nt1_nt2 !=) (nx1_nx2 ==)  ==> far_1
 { pos=  0.47,  neg=  0.28,  cf=  0.621
  p=49, n=291, tp=104, tn=1040 }

(nx1_nx2 ==) (fun1_fun2 !=)  ==> far_1
 { pos=  0.47,  neg=  0.28,  cf=  0.621
  p=49, n=291, tp=104, tn=1040 }

(nx1_nx2 ==) (ni1_ni2 !=)  ==> far_1
 { pos=  0.47,  neg=  0.28,  cf=  0.621
  p=49, n=291, tp=104, tn=1040 }

(nx1_nx2 !=)  ==> near_s
```

```
 { pos=  0.68,  neg=  0.40,  cf=  0.624
  p=55, n=424, tp=81, tn=1063 }

(fun1_fun2 ==)  ==> way_out
 { pos=  0.38,  neg=  0.18,  cf=  0.669
  p=233, n=97, tp=619, tn=525 }

(ni1_ni2 ==)  ==> way_out
 { pos=  0.46,  neg=  0.21,  cf=  0.688
  p=287, n=110, tp=619, tn=525 }

(nt1_nt2 ==)  ==> way_out
 { pos=  0.46,  neg=  0.21,  cf=  0.686
  p=283, n=109, tp=619, tn=525 }

(w1_w2 ==)  ==> way_out
 { pos=  0.46,  neg=  0.21,  cf=  0.686
  p=283, n=109, tp=619, tn=525 }

(h1_h2 ==)  ==> way_out
 { pos=  0.46,  neg=  0.21,  cf=  0.686
  p=283, n=109, tp=619, tn=525 }

(com_src yes)  ==> way_out
 { pos=  0.09,  neg=  0.04,  cf=  0.687
  p=56, n=21, tp=619, tn=525 }

(com_dest yes)  ==> way_out
 { pos=  0.08,  neg=  0.02,  cf=  0.772
  p=50, n=12, tp=619, tn=525 }

(com_src no) (w1_w2 !=) (nx1_nx2 ==)  ==> close_1
```

```
 { pos=  0.35,  neg=  0.22,  cf=  0.611
  p=43, n=220, tp=123, tn=1021 }

(com_src no) (w1_w2 !=) (ni1_ni2 !=)  ==> close_1
 { pos=  0.78,  neg=  0.50,  cf=  0.608
  p=96, n=507, tp=123, tn=1021 }

(com_src no) (h1_h2 !=) (nx1_nx2 ==)  ==> close_1
 { pos=  0.35,  neg=  0.22,  cf=  0.611
  p=43, n=220, tp=123, tn=1021 }

(com_src no) (h1_h2 !=) (ni1_ni2 !=)  ==> close_1
 { pos=  0.78,  neg=  0.50,  cf=  0.608
  p=96, n=507, tp=123, tn=1021 }

(a1_a2 !=) (nx1_nx2 ==)  ==> far_1
 { pos=  0.34,  neg=  0.16,  cf=  0.662
  p=35, n=171, tp=104, tn=1040 }

(com_src no) (nt1_nt2 !=) (nx1_nx2 ==)  ==> close_1
 { pos=  0.35,  neg=  0.22,  cf=  0.611
  p=43, n=220, tp=123, tn=1021 }

(com_src no) (nt1_nt2 !=) (ni1_ni2 !=)  ==> close_1
 { pos=  0.78,  neg=  0.50,  cf=  0.608
  p=96, n=507, tp=123, tn=1021 }

(com_src no) (w1_w2 !=) (ni1_ni2 !=)  ==> near_s
 { pos=  0.81,  neg=  0.51,  cf=  0.613
  p=66, n=537, tp=81, tn=1063 }
```

When rule_learner is finished, the rules listed above will also be listed in the file adder.rules. This file is in a format that the inference engine tool (ie) can use. This list of rules represents the input-to-output relationships that resulted from using artistII on a set of training designs including a 4-bit, 8-bit, 12-bit, 16-bit, and 32-bit adder. When the tool ie applies these rules to a target design, the resultant output is a list of predicted layout concepts for that design. These concepts make it possible to derive area and delay estimates for the target design without producing an actual layout.

Overall, once rule_learner produces the set of rules, the tool called match can pre-processes the target design, call ie to apply the rules and produce the predicted concepts list, and then use this list to estimate area and delay.

FILES

/usr/users/software/est/bin
/usr/users/software/est/src
/usr/users/software/est/learn
/usr/users/software/est/learn/train

SEE ALSO

artistII(1), match(1), ie(1), clusterII(1), glue(5)

AUTHORS

Don S. Gelosh

APPENDIX E

APPENDIX E

NAME

match - a training set builder and area and delay estimator for microprocessor ALU components based on CMOS two dimensional, gate matrix type layouts

SYNOPSIS

match [-t] [-smodel -rrows] [file]

DESCRIPTION

match is a VLSI CAD tool that operates on microprocessor ALU designs. match performs two functions. One function is building training sets based on these designs that train a machine learning tool. This learning tool, called rule_learner, produces a set of rules that describe the input-to-output relationships resulting from using the layout tool, artistII. The other function is estimating area and delay for these designs using this set of rules. In the training mode, match uses as its input a flattened GLUE description of the training design (file.fla), a layout description file (file.grp) that describes the actual layout produced by artistII for the design, and a node description file (anodes.phy) that describes the physical characteristics of all nodes handled by artistII. match outputs a training data file (file.data) and a domain description file (file.names) that are used by rule_learner in producing the set of rules. In the estimator mode, match uses as its input the set of rules (model.rules) produced by the rule_learner program and outputs to the screen area and delay estimates. match also produces a file describing an estimated layout (file.clu) and a file des-

cribing the critical path (file.crit).

OPTIONS

-t This argument puts the match tool into training mode.

-smodel
This argument puts the match tool into rule application and estimating mode. It also tells match which model (set of rules) to use.

-rrows
This argument tells match how many rows to use in the estimated layout (estimator mode only.)

EXAMPLE (training mode)

To use match to produce a training set using a 4-bit adder training design (stored in add4.fla) and its corresponding artistII layout description (add4.grp) issue the command:

```
match -t add4
```

match will output the following information to the screen as well as create the files add4.data and add4.names:

```
Filename: add4
Begin processing flattened description file.
Begin processing layout description file.
Layout: 8 rows by 8 cols.  Number of nodes: 43.
Generate adjacency matrix.
Transform distance matrices.
Produce data files.
```

This process must be repeated for the remaining training designs. After all training designs for a particular training set have been processed by match, all of the individual training data files (file.data) can be concantenated into one large file which constitutes the training set. This large data file along with any one of the domain description files (file.name) is then used by rule_learner to produce the set of rules that model the input-to-output relationships from artistII.

EXAMPLE (estimator mode)

To use match to apply the rules (stored in adder.rules) and estimate area and delay for a 16-bit adder target design (stored in add16.fla) with 8 rows in the estimated layout issue the command:

```
match -sadder -r8 add16
```

match will output the following information to the screen as well as create the estimated layout (add16.clu) and critical path (add16.crit) description files:

```
Filename: add16
Begin processing flattened description file.
Layout: 8 rows by 32 cols.  Number of nodes: 199.
Generate adjacency matrix.
Transform distance matrices.
Produce data files.
Apply layout tool model (set of rules):
Read 256 cases (11 attributes) from adder.test
Read 31 rules from adder.rules
```

```
Find height and width estimates:
Estimated Height = 944 lambda.
Estimated Width  = 1492 lambda.
Find critical path delay.
Predicted Delay[0] = 58.92 ns.
Predicted Delay[1] = 90.37 ns.
```

The match program first pre-processes the target design into a data file (add16.data). Then it calls a tool named ie (inference engine) which is part of the rule_learner system. The ie tool applies the rules (stored in adder.rules) to the data file. As a result from applying the rules, the ie tool produces an intermediate file (adder.pred) which holds the predicted layout information. match uses the information in this file to build an estimated layout for estimating area and to calculate all of the node-to-node delays for estimating the critical path delay.

FILES

/usr/users/software/est/bin
/usr/users/software/est/src
/usr/users/software/est/learn
/usr/users/software/est/learn/train
/usr/users/software/est/target

SEE ALSO

artistII(1), rule_learner(1), ie(1), clusterII(1), glue(5)

AUTHORS

Don S. Gelosh

APPENDIX F

APPENDIX F

NAME

ie - a simple inference engine used to apply the rules learned by the machine learning tool called rule_learner.

SYNOPSIS

ie [file]

DESCRIPTION

ie is a simple voting inference engine which takes the rules learned by the rule_learner tool (file.rules), and applies them to the examples (file.test). The output is a file (file.pred) which lists each example and its predicted classification.

EXAMPLE

For an example that uses ie, see the man page for the match(1) tool.

NOTE

The current version of ie was modified to operate in a computer aided design (CAD) domain application where the machine learning tool rule_learner is first used to model the input-to-output relationships that result from using layout tools. These relationships are represented by a set of rules. When these rules are applied to a target design by running ie, the result is a list of predicted layout concepts. These concepts are used to estimate area and delay for the target design without producing an actual layout.

Another tool called match pre-processes the target design, calls ie to apply the rules and produce the list of predicted layout concepts, and uses this list to estimate area and delay. Therefore, in this application, ie is not a stand-alone tool.

FILES

/usr/users/software/est/bin
/usr/users/software/est/src
/usr/users/software/est/learn
/usr/users/software/est/learn/train

SEE ALSO

artistII(1), match(1), rule_learner(1), clusterII(1), glue(5)

AUTHORS

Don S. Gelosh

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] McFarland, M.C., Parker, A.C., and Camposano, R., "Tutorial on High-level Synthesis," *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 1988, pp. 330-336.

[2] McFarland, M.C., Parker, A.C., and Camposano, R., "The High-Level Synthesis of Digital Systems," *Proceedings of the IEEE*, Vol. 78, No. 2 (February 1990), pp. 301-317.

[3] Knapp, D.W., "An Interactive Tool for Register-Level Structure Optimization," *Proceedings of the 26th Design Automation Conference*, pp. 598-601, 1989.

[4] Knapp, D.W., "Manual Rescheduling and Incremental Repair of Register-Level Datapaths," *Proceedings ICCAD-89*, pp. 58-61, 1989.

[5] Knapp, D.W., "FASOLT: A Program for Feedback-Driven Data-Path Optimization," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 6, (June 1992), pp. 677-695.

[6] Kurdahi, F.J., "Area Estimation of VLSI Circuits" (Ph.D. thesis, University of Southern California, 1987).

[7] Kurdahi, F.J. and Parker, A.C., "Techniques for Area Estimation of VLSI Layouts," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 1, (Jan. 1989), pp. 81-92.

[8] Jain, Rajiv, Kucukcakar, Kayhan, and Parker, Alice C., "Experience with the ADAM Synthesis System," *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989.

[9] Jain, R., Parker, A.C., and Park, N., "Predicting System-Level Area and Delay for Pipelined and Nonpipelined Designs," *IEEE Transactions on Computer Aided Design*, August 1992.

[10] Breuer, M., "Min-Cut Placement," *Journal of Design Automation and Fault Tolerant Computing*, pp. 343-362, 1977.

[11] Kernighan, B. and Lin, S., "An Efficient Procedure for Partitioning Graphs," *Bell System Technical Journal*, pp. 291-307, 1970.

[12] Lauther, U., "A Min-cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation," *Journal of Digital Systems*, vol.4, no. 1, pp. 21-34, 1980.

[13] Kirkpatrick, S., Gelatt, C.D. Jr., and Veechi, M.P., "Optimization by Simulated Annealing," *Science*, V220-4598, pp. 671-680, May 1983.

[14] Rutenbar, R.A., "Simulated Annealing Algorithms: An Overview," *IEEE Circuits and Devices*, pp. 19-26, 1989.

[15] Sechen, C. and Lee, K.W., "An Improved Simulated Annealing Algorithm for Row-Based Placement," *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[16] Hakimi, S., "Steiner's Problem in Graphs and its Implications," *Networks 1*, pp. 113-133, 1971.

[17] Kang, S., "Linear Ordering and Application to Placement," *Proceedings of the 20th IEEE/ACM Design Automation Conference*, pp. 457-464, 1983.

[18] Schuler, D. and Ulrich, E., "Clustering and Linear Placement," *Proceedings of the 9th Design Automation Workshop*, pp. 50-56, 1972.

[19] Ramachandran, C. and Kurdahi, F.J., "Combined Topological and Functionality Based Delay Estimation Using a Layout-Driven Approach for High Level Applications," *Proceedings of the European Design Automation Conference*, 1992.

[20] Ramachandran, C. and Kurdahi, F.J., "TELE: A Timing Evaluator Using Layout Estimation for High Level Applications," *Proceedings of the European Design Automation Conference*, 1992.

[21] Landman, B. and Russo, R., "On a Pin Versus Block Relationship for Partition of Logic Graphs," *IEEE Transactions on Computers*, vol. C-20, pg. 1469, 1971.

[22] Ramachandran, C. and Kurdahi, F.J., "A Combined Topological and Functionality Based Delay Estimation Using a Layout-Driven Approach for High Level Applications," *Proceedings of the European Design Automation Conference*, 1992.

[23] Ramachandran, C., Kurdahi, F.J., Gajski, D.D., Wu, A.C.-H., and Chaiyakul, V., "Accurate Layout Area and Delay Modeling for System Level Design," *International Conference on Computer Aided Design*, 1992.

[24] Penfield, P., Jr., and Rubenstein, J., "Signal Delay in RC Tree Networks," *Proceedings of the 18th Design Automation Conference*, 1981.

[25] Weste, N. and Eshraghian, K., *Principles of CMOS VLSI Design: A Systems Perspective*, (Reading: Addison-Wesley, 1988).

[26] Nourani, M. and Papachristou, C., "A Layout Estimation Algorithm for RTL Datapaths," *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993, pp. 285-291.

[27] Kim, S., "CMOS VLSI Layout Synthesis for Circuit Performance" (Ph.D. thesis, The Pennsylvania State University, 1992).

[28] Irwin, M.J. and Owens, R.M., "A Comparison of Four Two-Dimensional Gate Matrix Layout Tools," *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 698-701, 1989.

[29] Kim, S. and Owens, R.M., "A Two Dimensional Gate Matrix Module Generator," *Proceedings of the 3rd Physical Design Workshop, Laurel Highlands*, PA, May 1991.

[30] Sechen, C., *VLSI Placement and Global Routing Using Simulated Annealing*, (Boston: Kluwer Academic, 1988).

[31] Kim, S., Owens, R.M., and Irwin, M.J., "A Module Generator for High Performance CMOS Circuits," *Proceedings of the 1992 IEEE International Symposium on Circuits and Systems*, vol. 3, pp. 1266-1269, San Diego, CA, May 1992.

[32] Kim, S., Owens, R.M., and Irwin, M.J., "Experiments with a Performance Driven Module Generator," *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 687-690, 1992.

[33] Kramer, M.R., and van Leeuwen, J., "The Complexity of Wire Routing and Finding Minimum Area Layouts for Arbitrary VLSI Circuits," *Advances in Computing Research, vol. 2, VLSI Theory*, F.P. Preparata, ed., Jai Press Inc., Greenwich, CT, pp129-146, 1984.

[34] Rivest, R.L. and Fiduccia, C.M., "A 'Greedy' Channel Router", *Proceedings of the 19th Design Automation Conference*, 1982.

[35] Szymanski, T.G., "Dogleg Channel Routing is NP-complete," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. CAD-4, pp. 31-41, 1985.

[36] Garey, M. and Johnson, D., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, (San Francisco: Freeman, 1979).

[37] Romeo, F. and Sangiovanni-Vincentelli, A., "Probabilistic Hill Climbing Algorithms: Properties and Applications," *1985 Chapel Hill Conference on Very Large Scale Integration*, pp. 393-417, Ed. H.Fuchs, (Rockville: Computer Science Press, 1985).

[38] Irwin, M.J. and Owens, R.M., "An Overview of the Penn State Design System," *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pp. 516-522, 1987.

[39] Hou, P-P, and Owens, R.M., "GLUE," *UNIX System Man Pages*, 1990.

[40] Levitan, S.P., Martello, A.R., Irwin, M.J., and Owens, R.M., "Using VHDL as a Language for Synthesis of CMOS VLSI Circuits," *Proceedings of the 9th International Symposium on Computer Hardware Description Languages*, pp. 331-346, 1989.

[41] Owens, R.M. and Irwin, M.J., "Exploiting Gate Clustering in VLSI Layout," *Technical Report CS-88-09*, Computer Science, The Pennsylvania State University, University Park, PA, 1988.

[42] Hou, P-P, and Owens, R.M., "ArtistII," *UNIX System Man Pages*, 1990.

[43] Ousterhout, J., Hamachi, G., Mayo, R., Scott, W., and Taylor, G., "Magic: A VLSI Layout System," *Proceedings of the 21st Design Automation Conference*, pg. 152, 1984.

[44] Scott, W.S., Mayo, R.N., Hamachi, G., and Ousterhout, J.K., "1986 VLSI Tools: Still More Works by the Original Artists," *Technical Report*, Computer Sciences Division (EECS) University of California, Berkeley, CA, 1985.

[45] Sechen, C., and Sangiovanni-Vincentelli, A., "TimberWolf3.2: A New Standard Cell Placement and Global Routing Package," *Proceedings of the 23rd Design Automation Conference*, pp. 432-439, 1986.

[46] Sechen, C., and Sangiovanni-Vincentelli, A., "The TimberWolf Placement and Routing Package," *Proceedings of the Custom Integrated Circuits Conference*, 1984.

[47] Goto, S. and Kuh, E., "An Approach to the Two-Dimensional Placement Problem in Circuit Layout," *IEEE Transactions on Circuits and Systems*, pp. 208, 1978.

[48] Stevens, J., "Fast Heuristic Techniques for Placing and Wiring Printed Circuit Boards" (Ph.D. thesis, University of Illinois, 1972).

[49] Buchanan, B. and Shortliffe, E. *Rule-Based Expert Systems*, (Reading: Addison-Wesley, 1984) pp. 247-262.

[50] Michalski, Ryszard S., Carbonell, Jaime G., and MItchell, Tom M., eds. *Machine Learning - An Artificial Intelligence Approach*, (Palo Alto: Tioga, 1983).

[51] Tanimoto, S.L., *The Elements of Artificial Intelligence Using Common Lisp*, (New York: Computer Science Press, 1990).

[52] Weiss, Sholom M. and Kulikowski, Casimir A., *Computer Systems That Learn*, (San Mateo: Morgan Kaufmann, 1991).

[53] Provost, F.J., "Policies for the Selection of Bias in Inductive Machine Learning" (Ph.D. Thesis, University of Pittsburgh, 1992).

[54] Provost, F.J., Buchanan, B.G., Clearwater, S.H., Lee, Y., Leng, B., "Machine Learning for Exploratory Mining," (To be published in Machine Learning).

[55] Samuel, A.L., "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, no. 3, pp. 211-229, 1959.

[56] Hecht-Neilsen, R., *Neurocomputing*, (Reading: Addison-Wesley, 1990) pg 115.

[57] Khanna, T., *Foundations of Neural Networks*, (Reading: Addison-Wesley, 1990).

[58] Rosenblatt, F., "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychology Review*, Vol 65, pp. 386-408, 1958.

[59] Manber, U., *Introduction to Algorithms*, (Reading: Addison-Wesley, 1989).

[60] Buchanan, B. and Mitchell, T., "Model-directed Learning of Production Rules," in D.A. Waterman and F. Hayes-Roth (eds.), *Pattern Directed Inference Systems*, (New York: Academic Press, 1978), pp. 297-312.

[61] Camposano, R. and Tablet, R.M., "Design Representation for the Synthesis of Behavioral VHDL Models," *Proceedings of the 9th International Conference on Computer Hardware Description Languages*, 1989.

[62] Camposano, R., Saunders, L.F., and Tabet, R.M., "VHDL as Input for High-Level Synthesis," *IEEE Design and Test of Computers*, March 1991, pp. 43-49.

[63] *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1987, IEEE Computer Society Press, Los Alamitos, CA, 1987.

[64] Hou, P-P, and Owens, R.M., "IVF2GLUE," *UNIX System Man Pages*, 1990.

[65] Hsieh, Yee-Wing, "Architectural Synthesis Via VHDL" (M.S. thesis, the University of Pittsburgh, 1992).

[66] Terman, C.J. "Simulation Tools for Digital LSI Design" (Ph.D. thesis MIT Laboratory of Technology for Computer Science, 1983).

[67] Personal communication with Steven P. Levitan, Assistant Professor, Department of Electrical Engineering, University of Pittsburgh, PA., September 6, 1993.

[68] Blaisdell, E.A., *Statistics in Practice*, (Fort Worth: Saunders College, 1993).

[69] Caulcutt, R., *Statistics in Research and Development*, (New York: Chapman and Hall, 1991).

[70] MCNC, Center for Microelectronic Systems Technologies, 3021 Cornwallis Road, P.O. Box 12889, Research Triangle Park, N.C. 27709.

[71] Kane, Gerry, and Heinrich, Joe, *MIPS RISC Architecture*, (Englewood Cliffs: Prentice Hall, 1992) pg. 1-1.

[72] Preas, Bryan, and Lorenzetti, Michael, eds. *Physical Design Automation of VLSI Systems*, (Menlo Park: Benjamin/Cummings, 1988).

[73] Setliff, D.E. and Rutenbar, R.A., *Automatic Programming Applied to VLSI CAD Software: A Case Study*, (Boston: Kluwer, 1990).

[74] Sherwani, Naveed A., *Algorithms for VLSI Physical Design Automation*, (Boston: Kluwer Academic, 1993).

[75] Modell, M.E., *Data Analysis, Data Modeling, and Classification*, (New York: McGraw-Hill, 1992).